IBM原厂“正规军”出品

DB2

设计、管理与性能优化艺术

王飞鹏 李玉明 朱志辉 王富国 编著

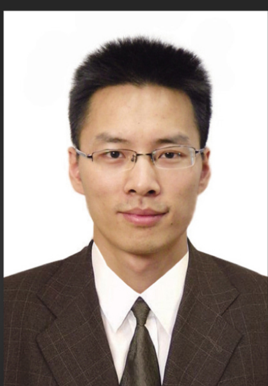
本书特色：

- IBM原厂性能优化专著
- DB2 pureScale设计与优化专著
- DB2数据仓库设计与优化专著
- 涵盖DB2 HADR灾备最佳实践
- 涵盖DB2 V10最新设计与优化技术
- 将Oracle与DB2的对比贯穿全书
- 融汇金融、电信、政府等行业实战案例
- 表达新颖独特、内容深入浅出
- DB2从业人员的案头必备之作

清华大学出版社

王飞鹏

IBM公司DB2资深顾问，首屈一指的数据库开发、维护、优化和架构专家。曾为电信、银行、中央部委、中国高铁等大型数据库项目做出了重要贡献。发表数据库论文12篇，拥有软件专利3项。每年通过大量咨询、讲座、培训等方式，为数据库人才更好地运用数据库技术做出了最大的努力。



李玉明

DB2数据库资深软件工程师。多年来一直从事DB2内核编译器和优化器的开发工作。擅长DB2应用系统的架构和开发、大型数据库性能优化，对DB2存储过程的开发和内部运行机制有很深的造诣。曾发表多篇数据库技术论文，拥有IBM DB2各项认证。

DB2 设计、管理与性能优化艺术

王飞鹏　李玉明　朱玉辉　王富国　编著

清华大学出版社

北　京

序一

DB2 数据库是 IBM 公司数据管理产品线上最知名也是最成功的产品，不仅在大型主机市场处于绝对领导地位，而且在开放式平台的影响力更是与日俱增，不断地有客户从 Oracle 迁移到 DB2。

说起 DB2 的悠久历史，实际上它起源于一篇具有划时代意义的论文。1970 年，IBM 公司的研究员埃德加·考特(Edgar F. Codd)发表了名为“大型共享数据库的关系模型”（*A Relational Model of Data for Large Shared Data Banks*）的论文，从而正式拉开了关系型数据库的大幕。从那时起，IBM 公司就一直在数据库这个领域深耕细作，持续创新，通过不断研发优秀的产品屹立于强手如林的科技界，表现在 DB2 研发上更是不断的推陈出新。下面是 DB2 发展史上具有重要里程碑意义的版本，它们一经推出就获得了业界的广泛好评。

1983 年，DB2 正式问世，被 IBM 大型机所专用，之所以命名为 DB2，是由于以前已经有了一个著名的层次型数据库产品 Information Management System（IMS）。

1992 年，IBM 将 DB2 带向了多种开放式平台，包括 Linux、Unix 以及 Windows 服务器，也简称为 DB2 LUW。

2002 年，IBM 发布了 DB2 V8，这个版本开始支持数据库分区特性（Data Partitioning Feature，DPF），从而通过非分享（Share Nothing）的 MPP 架构为数据仓库提供更强的伸缩性。

2006 年，IBM 发布了 DB2 V9，这个全新的 DB2 是第一款“天然”存储 XML 的关系型数据库。

2008 年，IBM 发布了 DB2 V9.7，提供了 Oracle 兼容特性，这个版本完全支持 Oracle 语法，从而大大方便了应用和数据库迁移。

2009 年，IBM 发布了 DB2 V9.8，增加了 DB2 pureScale 特性，该特性利用了 z/OS 大型机上经过验证的 DB2 数据库集群技术，在开放平台上实现了共享磁盘（Share Disk）架构，以满足高吞吐量 OLTP 应用对高可用性、扩展性和负载均衡的需要。

2012 年 4 月，IBM 发布了 DB2 V10，该版本提供了多时态表、多表星型连接、行列访问控制（RCAC）、多温度存储、自适应压缩等特性。在这个版本中，DB2 pureScale 技术得到了进一步完善，从而获得了很多行业客户的高度认可，目前 DB2 pureScale 已经在中国金融、航空、烟草等行业的生产库上得到了成功实施。

2012 年 10 月，IBM 正式推出了数据库一体机 PureData。该产品不同于竞争对手的支持混合负载一体机，而是针对交易处理、分析和操作分析这三种应用场合分别提供不同的软硬件预配置解决方案。

时光荏苒，我加入 IBM 中国开发中心已经快 20 年了，在这里非常荣幸见证了中国工程师 DB2 研发实力的快速提高。从刚开始接一些简单的开发工作，到现在参与 DB2 最新的核心技术

开发；从刚开始国外团队对中国工程师工作能力的半信半疑，到现在获得国外 DB2 团队的信任和认可，这说明中国开发中心在 IBM 全球的研发地位变得越来越重要了。当然，除了研发最新的 DB2 核心技术外，中国开发中心还作为培养中国 DB2 人才的“黄埔军校”，培养了一大批技术支持和服务客户的优秀工程师。

成绩是非常突出的，但是存在的一些问题也毋庸讳言。在数据库用户中，精通 Oracle 的人非常多，但是精通 DB2 开发、维护和优化的人就相对比较少了，这在一定程度上影响了 DB2 产品的使用。其实，说到底这是 DB2 生态系统的问题。作为原厂来说，IBM 公司是非常愿意通过各种方式和途径推广 DB2 产品技术的，例如通过和 db2china 社区的紧密合作，定期由来自中国研发中心的资深工程师在 db2china 社区上轮值来解答用户的棘手问题，中国开发中心也会和 db2china 社区定期举办 DB2 沙龙以更有效地传播 DB2 新技术，这些努力都取得了非常好的效果。

近几年来，我们还逐渐加大了对 IBM 客户和合作伙伴的支持力度。针对 DB2 客户，我们于 2011 年在大中华区成立了 DB2 客户顾问委员会，简称 DB2 CAC，目前来自银行、电信、保险、制造等不同行业的多个客户已经加入了这个委员会。这种方式加强了 DB2 实验室和客户之间的双向交流与合作，我们为客户安排了实验室顾问（Lab Advocate），向他们介绍 DB2 最新的技术发展，客户也可以把他们对 DB2 产品的需求提供给 DB2 开发团队，这种方式收到了非常显著的效果。另外，针对合作伙伴，我们今年成立了一个合作伙伴委员会（Business Partner Council）以帮助 IBM 重要合作伙伴在他们的业务系统中使用好 DB2。

实际上，除了 IBM 官方的积极推动外，类似《DB2 设计、管理与性能优化艺术》这样的 DB2 书籍也是一种非常有效的技术推广方式。这本书由来自 IBM 中国开发中心的 DB2 资深工程师编写，具备科学的性能优化知识体系，以大量实战案例为载体，不仅展示了 DB2 经典设计和优化方法学，还涵盖了 DB2 高级设计和优化技术，例如 DB2 HADR、DB2 pureScale 集群、数据仓库等。这对奋战在数据库应用开发与优化一线的 DB2 用户来说，可以说是找到了一把进入 DB2 技术圣殿的金钥匙，希望广大读者能从中获益。

IBM 中国开发中心 DB2 开发与客户支持总监

干毅民

2013 年 7 月 1 日

序二

去国离家十四载后，2009 年，当我再次从 IBM 多伦多实验室归来凝视上海这座不夜城时，不由地感慨世界真的变小了、变平了，中国作为地球村的一员，正以惊人的速度发展变化着，而我的工作也一样在发展变化着。从加拿大回国后于 2010 年正式加入 IBM 中国软件开发实验室，由之前的 DB2 数据管理工作，转而开始从事数据治理（Data Governance）方面的工作。

过去的十年是企业的 IT 系统数据量高速膨胀的时期，“大数据”时代已悄然来临：每天，遍布世界各个角落的传感器、移动设备、在线交易和社交网络生成上百万兆字节的数据；每个月，人类发布 10 亿条 Twitter 信息和 300 亿条 Facebook 信息。据统计，全球 90% 的数据都是在过去两年中生成的。

“大数据”时代必然会产生新的“大数据”问题。哪些数据是可信的？哪些数据需要进行清洗？如何从海量数据中获得业务洞察力，从而指导商业决策？如何确保新录入的信息不会产生冗余？如何以可复用的方式发布可信任的信息？如何使如此庞大的数据真正变成对企业有价值的信息？这些海量的、分散在不同角落的数据带来了资源利用的复杂和管理的困难等问题。

以上所述各种问题，最终的解决办法就是要靠数据治理。（在很多时候我们用一个更为精确的概念——信息治理）数据治理是专注于将数据作为企业的商业资产进行应用和管理的一套管理机制。良好的数据治理能够消除数据的不一致性，提高组织数据质量，建立规范的数据应用标准，实现数据广泛共享，并能够将数据作为企业的宝贵资产应用于业务、管理、战略决策中，发挥数据资产的最大商业价值。

这几年，数据治理在国内的研究和应用都取得了一定进展，但是也面临观念上和实践上的双重挑战。从观念上看，国内很多人认为数据治理只是 IT 部门的责任，只把数据治理当成软件，并未真正意识到业务、数据和软件之间的关系，也就不能从整体上将数据作为企业资产来看待。从实践上看，很多企业做了数据质量检查，做了数据归档，做了数据安全，但缺乏一个完整的体系来将各个部分串联起来，也就是说，缺乏将这些领域组织起来的方法论。

为了帮助企业更好地管理数据资产，应用大数据时代信息治理的挑战，IBM 推出了全新的信息管理和业务分析产品，并提供技术资源，致力于为企业及机构提供大数据分析、信息整合、主数据管理等数据治理解决方案。以下都是基于 IBM 数据治理方案的优秀软件。

IBM InfoSphere BigInsights 是 IBM 大数据平台的核心产品之一，它是一款以 Hadoop 为基础的、对海量数据进行存储、管理和分析的企业级平台。可以在 30 分钟内安装完毕并投入运行，可用来管理企业各种数据，比如大量来自社交网络、移动设备和传感器等不同来源的非结构化数据，并对这些数据进行深度挖掘和分析。

IBM InfoSphere Information Server 系列软件支持将大数据作为来源和目标进行整合，并凭借其成熟可靠的性能和并行引擎，提供大数据所需的强大可扩展性，包括元数据管理、数据清洗、

数据质量治理及分析、数据自动化分析、数据抽取转换加载、数据集成、监控与报告等一整套软件组成的企业级信息平台。通过它可分析、清洗和整合异构源中的数据信息，并且把经过分析、清理后的可信任信息以可复用的方式提供给用户，同时也对新录入的信息进行实时的数据清洗操作，保证新录入信息的正确性。

IBM InfoSphere Master Data Management 软件，其最核心的任务是导出企业的关键业务数据，也是绝对真实的数据。主数据管理旨在从企业的多个业务系统中整合最核心的需要共享的数据，集中进行数据清洗，并以服务的方式把统一、完整、准确的主数据分发给企业内的操作型应用和分析型应用，包括业务系统、业务流程和决策支持系统等。使用户更深入地理解生产链条上的各个要素——客户、产品、供应商、员工等之间的关系，为进一步分析和决策做重要支撑。

IBM InfoSphere Stream 是 IBM 大数据平台中专门针对快速产生的如流水般不间断的海量数据流的处理平台。它是一个支持开发和部署的应用程序平台。能够持续快速地分析实时产生的各种各样的海量流数据。具有低延迟、实时响应、跨多个数据流进行分析的特点，尤其适用于对响应时间有较高要求的应用，例如欺诈检测、网络管理，能很好地解决企业处理数据量大、存储成本高的问题。通过直接对数据进行分析，无须存储，从而实现对有价值数据进行深入分析的可能。能够在大规模的集群环境中并行、高性能地处理流数据，并具有近似于线性的可扩展性，是帮助企业处理实时化的海量流数据分析的好帮手。

谈了不少数据治理，那么它和 DB2 数据管理有什么不同？我想分享一下我的观点。如果说从事数据质量管理、主数据管理等数据治理工作的人是数据巨轮的船长，他们平时的工作就是站在舰桥上，穿着带金色肩章的白制服，用双筒望远镜了望远方以把握方向，那么从事 DB2 相关工作的 DBA 就是在轮机舱工作的船员，船长和船员双方共同为数据巨轮的稳定运行发挥重要作用。但是，在实际工作中，我经常发现这样的事情：当舰桥上传来船长焦急的指令，命令 DBA 加快数据流动的速度时，DBA 由于缺乏驾驭 DB2 的优化技巧，只能回答说，“DB2 引擎遇到性能瓶颈了，船长！”。虽然船长的指令很及时，但遇到这样的船员，数据巨轮恐怕也运转不佳了。

DB2 船员的实际水平制约了船长对数据巨轮的驾驭能力。本书就是一本供 DB2 船员行驶的实战手册，希望广大船员能真正理解和掌握书中讲述的 DB2 设计与性能优化艺术，当收到船长要求加速数据流动的命令时，可以让 DB2 引擎运行得更好，这样数据巨轮就能畅通无阻得行驶下去。

IBM 中国开发中心信息管理产品开发部

洪桦 资深主管经理

2013 年 6 月 8 日

序三

IBM 百年华诞，在 2011 年夺目九界，璀璨全球。创新，这是 IBM 能屹立于强手如林的科技界的关键，使得她能够适应科技时代发展的需要，不断创新求变，从而把握时代的脉搏，解决今天以及未来企业遇到或可能遇到的重大挑战。

进入 21 世纪，IBM 与其众多对手们不约而同地投入到了信息时代的竞争。随着信息技术的迅猛发展，作为其核心组成部分的数据之战已成为了 21 世纪“竞争”的新内涵，而作为承载、处理和加工这些数据的数据库软件行业就不可避免地成为了主战场。

为了满足客户各种需求，大家竞争的对象是极富生命力的数据。在数据的整个生命历程中，它会经历设计、开发、部署、运营、优化和治理的不同阶段，并且这不是一次性的过程，而是通过周期性的迭代方式，来发挥数据更大的价值。任何一家企业拥有了对数据强大的管控和支配，它就会在 21 世纪数据之战中立于不败地位，甚至引领信息时代的发展。

IBM 正在着手于实现一个战略计划，提供一个集成的模块化数据管理环境，帮助企业更高效准确地管理整个数据生命周期（从需求到报废）。我们将这个过程称为“集成数据管理”，管理数据生命周期的每个环节，并能够支持各种主流厂商提供的数据管理技术，这包括 DB2、Informix、Oracle 等。

本文中提到的 IBM Optim Tools，就是应运而生的这样一个工具集，它除了提供对数据库基本的管理和开发功能外，还提供了强大的 DB2 监控和优化功能。IBM Optim Tools 最大的优势就在于对 DB2 数据库全面的支持，能够及时地反映并紧跟上 DB2 数据库的发展和更新，例如对 DB2 数据仓库和 DB2 pureScale 的支持。

本书的作者都有非常丰富的数据库管理和优化经验，使得本书具有极佳的实践性和可操作性，相信能为广大的数据库用户提供前所未有的帮助。

IBM 中国开发中心信息管理产品开发部
资深经理 孙冰江
2013 年 6 月 28 日写于北京

序四

我和飞鹏在几年前的一次InfoSphere培训中结识，基于对DB2的共同爱好，我们经常探讨关于DB2的各种技术，受益颇丰。如今飞鹏的舞动DB2系列专著已经陆续出版上市，作为已经阅读过舞动DB2系列书籍的读者，我想说，这套专著的理论水平毋庸置疑，实际案例更是经验和智慧的结晶，对于使用DB2的各个层次的读者都会有很大帮助。

这次，新书《DB2设计与性能优化艺术》即将上市了，我受飞鹏之邀为本书作序，感到十分荣幸。

先讲讲我和数据库的故事。我从1992年开始使用数据库，最初是用FOXBASE开发了一个工资管理系统。1995年开始基于DB2 for AS400的银行应用开发。1997年开始基于SYBASE的证券应用开发。1998年开始从一名开发者转向系统工程师，开始了中间件和数据库方面的技术支持。1999年获得DB2 V5 DBA认证。1999年~2006年，先后做过SYBASE、DB2、Oracle数据库的规划设计、系统管理、故障排除、性能优化、开发指导等工作。2007年开始主要关注和研究IBM软件，提供主流数据库、数据仓库、数据迁移、灾难备份的咨询及培训服务。到目前为止，我拥有DB2从版本5到版本10的DBA认证、DB2 版本9的高级DBA认证、Oracle 11G DBA OCP认证、Informix 10和11的DBA认证以及IBM的InfoSphere及Tivoli的各种认证40余个。目前担任IBM官方认证讲师，负责十多项产品的培训工作。

虽然我做过各种类型的数据库工作，但是数据库的设计与优化无疑是一项很重要且关键的工作。说起数据库优化，先给大家讲一个故事。华佗兄弟三人都精通医术，有一天一个人问华佗，你们家里兄弟三人谁的医术最高，这时华佗说：我们家大哥的医术最高明，其次是我的二哥，医术最差的就是我了。提问者十分不解地问：所有人都知道你是天下最有名、医术最高的人了，怎么还有比你医术更高明的人？于是华佗说了一段非常耐人寻味的话："我大哥治病是在人们尚未察觉身体有病的时候为人们医治的，人们对他医术不甚了解。我二哥治病是在人们开始发病的时候通过望闻问切，开处方医治病人的，人们只是对他有所了解。我看病是在病人的病情非常严重的时候，才给病人下药，所以人们认为我能够让人起死回生，因此我最有名气，但是论医术水平我与我的兄长差距很大呀！"其实DBA就像是数据库的医生，数据库的性能优化也有三个阶段：第一个阶段是，数据库性能出现严重问题时去诊断分析、提出整改方案，即使能起死回生，但由于之前的规划设计缺陷很难达到持续稳定健康的运行；第二阶段是，在数据库运维过程及时发现问题解决问题，但同样也受限于之前的规划设计；第三个阶段是，在数据库规划设计阶段就开始考虑性能问题，这是性能优化的最高境界。所以设计与性能优化是分不开的，这也是本书所关注和阐述的。

通过十多年的数据库经验，强烈地感觉到数据库设计与性能优化需要了解数据库之外的很多相关知识，比如：操作系统、存储、网络、中间件，这些知识是相互关联和影响的，既而成

为一个既独立又统一的系统，理解和掌握的越多，在性能优化时考虑得越全面。DBA如同一个剑客，借用电影《英雄》的台词：“剑法有三种境界：第一种，手中有剑、心中却无剑，主要练就的是一招一式；第二，手中有剑、心中有剑，所谓人剑合一，练就的是剑气；第三，手中无剑、心中也无剑，是一种至大则空的平和。第三种被称为剑法的最高境界。” 第一种，手中有剑、心中却无剑：这种境界的DBA知道很多DB2的命令及用法，但对这些命令及用法的使用场景是否匹配合适则无所适从。第二种，手中有剑，心中有剑：知道很多命令及用法，也知道这些命令及用法的适用场景，也能做到活学活用。第三种，手中无剑、心中也无剑：不再使用现成的某某高手写的安装指南、运维指南等，不再仅关注于数据库，而更多地是利用DB2的信息中心，通过全面地系统感悟，做到IT系统为我服务，我来规划设计IT系统。到了这个境界，就不会纠结于要学习DB2还是Oracle，因为两者的很多东西都是相通的。希望本书能引导读者走向DBA的最高境界。

北京富通东方科技有限公司技术服务中心副总经理

张东焕

2013年7月15日写于北京

前 言

"三方演义"与性能优化

性能优化为什么这么难？

一个 IT 系统建设的过程，其实就是"三方演义"的过程。这里的三方，就是客户方、开发商和 IBM 原厂，通常是客户提需求，开发商负责应用设计和开发，IBM 提供软硬件产品，系统完成开发后，由客户或者第三方公司运行维护。一个 IT 系统的好坏，取决于这三方能否紧密合作，能否发挥各自独特的优势，从而实现 1+1+1>3 的效果。

类似于 IT 系统建设，一个上规模的性能优化项目，也是"三方演义"的互动过程。我经历过一些烂尾项目，这种项目在启动的时候，大家坐在一起指点江山，高谈政治，拍脑袋做决定，当出现性能问题的时候非常浮躁，不能冷静客观地分析问题和解决问题，而是叫喊着用更高档的硬件，缺乏严谨、务实、执着和求真的精神，缺乏对性能优化项目最起码的敬畏感。

首先是客户方。钱是客户出的，客户是真正的甲方，所以客户总是最强势，客户领导经常在会议上发号施令，当遇到问题的时候把开发商骂得一塌糊涂："看看你们写的垃圾代码，还想让我们购买更好的硬件？"或者骂原厂："你们说过 DB2 提供的自动维护特性不需要 DBA 过多干预，但为啥根本不是那么回事呢？"骂归骂，出了问题其实客户也是有责任的，你为啥不弯下身段，真正参与到系统的设计、开发和维护的整个过程呢？这样即使出了问题，心里也有底了呀。但是，现实中又很难，因为这和客户内部的体制有关，通常在一个项目的不同阶段，各个部门就像"铁路警察，各管一段"，开发部门只管开发，运维部门只负责上线维护，运维部门和开发部门是各自为战，当出了问题后，相互踢皮球，协调起来非常困难。例如，金融业的客户，一般拥有自己专门的开发部门和运维部门，两个部门独立，不存在上下级隶属关系，遇到困难后很容易相互扯皮。至于电信业和政府部门，出了系统性问题更难协调：电信业的客户，通常只负责系统维护工作，应用开发通常由第三方的开发商负责；政府客户，只负责需求、系统规划和管理工作，开发和维护都是由第三方公司承担的。针对客户方，我想强调的是，没有专业分工的局面是可怕的，但太过于精细的分工同样效率低下，更可怕的是很多企业采用政治挂帅的方式启动和管理项目，这是导致烂尾的重要根源之一。

接下来谈谈开发商。开发商是挨客户骂最多的了，但并不妨碍他们喜欢拍胸脯，啥都敢承诺，啥都敢做，毕竟有其独特优势。开发商有人力资源的优势，手中有大量的开发人员可供使用；开发商还有对业务理解的优势，毕竟开发商长期和行业客户在一起，积累了很多业务经验

和知识；通常开发商还有关系的优势，毕竟从客户那拿项目，是要和客户关系融洽、有信誉才行。但是，开发商也有技术硬伤，那就是对系统软件，包括硬件、数据库、中间件等的使用水平还停留在安装配置的层面，缺乏对其内部机制的了解，所以在架构规划、数据库优化、高可用性测试、可扩展性测试等方面技术上做不到甚至没有这方面的意识，更不可能实施了。

最后谈谈原厂。原厂的人员也会挨骂，也会被开发商拿来当挡箭牌，这不新鲜。原厂有自己的核心优势，那就是对自身产品的深入理解，以及跨金融、电信、政府等各行业应用所积累的实施优势，这必将在架构规划、数据库优化、扩展性测试等需要技术深度和经验的领域发挥重要作用，也可以为客户和开发人员在设计、开发和运维等工作上提供至关重要的技术支持。但是，原厂也有做的不够好的地方，其实也是体制上的原因，那就是以围绕产品的技术支持为主，如果在推动最新的技术和产品给客户的同时能愿意多倾听客户的真正需求，如果能积极主动参与到项目规划、设计、开发、测试和运维的整个阶段，那将对 IT 系统质量的提升发挥重要作用。

在这样一个“三方演义”的架构下，要想完成性能优化工作有时候超越了技术本身，它更多地需要设计开发人员和运维人员的紧密配合。比如，当我发现 SQL 语句的问题，需要开发人员支持和配合的时候，却被告知项目已经被开发商交付给客户了，没有办法找到开发人员了。有时候我即使找到了开发人员，他们却对 SQL 语句的最终运行环境一点都不关注，他们只是强调 SQL 语句是业务逻辑的需要。于是，我不得不找运维人员寻求支持，但是运维人员让我更吃惊，他们只关注主机、操作系统和数据库本身，对上层应用缺乏了解，也不关注上层 SQL 语句是怎么写的。

这可能是国内 IT 系统建设的一个缩影吧，所以在此我仅提出一条呼吁性质的建议：在客户的推动下，设计开发人员和运维人员定期举行技术交流，多交流硬件、数据库、中间件以及应用开发中遇到的性能痛点，拿出一套行之有效的办法在当前项目中试点并在其他项目中加以推广。

某银行性能优化的真实案例

2012 年 12 月 15 日，星期六，早上 8 点钟，本人衣冠楚楚，匆匆出门，打算去参加一个老同学的婚礼。刚上出租车，就接到了公司销售的电话，说是国内某银行客户的 DB2 数据库上线试运行后出现了非常严重的性能问题，希望我赶赴浙江宁波现场做一下性能调优工作。前几年如果周末接到这种电话，我内心肯定会抱怨一番的，周末了都不让人好好休息，不过经历的多了也就适应了，也慢慢练就了乱云飞渡仍从容的气度。老同学的婚礼该参加还是继续参加，该喝酒还是继续喝酒，该定去宁波的机票还是继续定。

我是 12 月 16 日晚上抵达宁波的，到了之后和客户领导郑总约了第二天也就是 12 月 17 日早上 9 点在现场召开会议。

第二天也就是星期一上午 8 点 50 分，我刚到了客户会议室门口，就听见一个人在满腔怒火地训斥，从声音可以听出来这个训人者就是昨天和我通电话的客户领导郑总。等我刚踏入会议室，发现场面非常隆重，里面坐了很多人，除了客户领导外，还有客户方 DBA 小李、第三方顾问公司的架构师老张、以及来自几家应用开发商的大队开发人员，可谓各路英豪齐聚一堂啊。作为原厂的唯一代表，本人孤独地坐在了角落里。会议几乎在争吵中度过，下面是我对大

家发言概要的整理。

郑总发言："这个项目是银监会重点督导的项目，具有重要的政治意义，如果这个项目在宁波试点成功了，那么可以推广到华东甚至是全国。但是，开发商在开发阶段就屡次拖延工期，这次上线试运行后，又暴露了严重的性能问题，业务人员没法使用，对此我是非常不满意的，需要拿出彻底的整改办法来。"

开发商代表发言："首先，接受领导的批判。但是，这个项目的业务需求改动了好几次，另外我们的队伍对 Weblogic 应用服务器和 Oracle 数据库非常熟悉，但是对 DB2 数据库开发和优化的技术积累非常薄弱啊，没有原厂的支持是不行的。"

郑总发言："原厂的工程师已经到了，王工，你终于出山了，昨天从北京过来的吧，你可是 DB2 领域的领军人物了，解决这个性能问题可以说是举手之劳啊……"

我发言："多谢大家的信任，初来乍到，我先了解一下情况再发表建议吧。"

客户方 DBA 小李发言（发言非常激烈）："这个应用上线试运行后，正常情况下还好，但是一旦达到了 1000 并发用户时，系统的平均响应时间从 1s 一下急剧增加到 4s 左右，业务人员根本就没法用，要知道现在的硬件可是 16 内核的双机 Power 740 了！"

开发商代表接着发言（已经快哭了）："现在双机 Power 740 配置太低了，难以满足性能需要，请按业务最高峰值配置硬件资源，使用最高档 Power 780，另外存储设备也要升级，用 IBM V7000！"

郑总发言："这算什么回事？不是国庆节前后刚从 x3850 升级到 Power 740 吗，当时你们可是告诉我，Power 740 肯定够用了，现在又要到 Power 780，你们真好意思说出口！"

开发商代表小声说："这个不会是 DB2 的 Bug 导致的吧……"

第三方顾问公司的架构师老张发言："现在是双机 Power 740，一台运行 Weblogic 应用服务器，一台运行 DB2，使用 HACMP 来实现 HA，当出现故障时，一台机器接管另外一台。现在 DB2 有新的技术了，可以用 DB2 pureScale 试试，如果还不行的话，可以考虑 DB2 一体机方案 pureData，应用跑上去肯定快几倍！"

我寻思着，首先感谢这个架构师对 IBM 新技术新产品的信任，不过他也有点太着急了，目前是响应时间慢的问题，不是事务吞吐量遇到瓶颈的问题，在没有真正分析客户的性能瓶颈之前，这样去硬推销 DB2 pureScale 或者 pureData 一体机，不仅不会给客户留下好感，有时候反而让客户对这些新技术或者新产品产生强烈的逆反心理。

会开到这个份上，其实已经陷入了僵局，我坐在角落里，大脑里反复出现郑总的话，经验告诉我，当客户遇到危难猛夸我的时候，其实已经把我推到了风口上了，因为终于有了可以堵枪眼的人了。当然，我也在考虑解决办法，内心也发出这样的感慨：相当一部分客户和开发商，什么都敢用，但是都用的不够专业，不够精细，这样一旦出现性能问题时，他们就只能高谈政治意义，随后习惯性地拍脑袋做决定，很难冷静客观地分析问题和解决问题。

快到中午了，郑总下午还有其他日常安排，开发商提出的用最高档 Power 780 的建议即将被一锤定音的时候，我赶紧发言，把我想说的用最快的速度说了出去："首先，我认为这个系统目前不应使用 Power 780 和 IBM V7000 存储，那是一种对资源的浪费，Power 740 跑这样的负载绰绰有余；其次，我认为 DB2 还有优化空间，我有办法调整一些参数让 DB2 把硬件的能力发挥出来；最后，也是最重要的，应用软件还有很大优化空间，至少在 1000 并发用户的情况

下，响应速度急剧变慢，就和应用有关。”

我说出上面的话后，整个会议室突然宁静了下来，开发商的人员面面相觑，客户领导这时反而笑了，笑的我有点发毛，他和蔼的问我：“王工，你有几成把握？这个项目可是非常紧急的，不能意气用事啊”，我内心其实也虚不过还是表面非常镇定地回答：“立足于现有的硬件环境，请给我 5 天时间吧，当我优化数据库和改造应用的时候，请 DBA 和开发人员配合我，谢谢！”

可能是外来的和尚会念经吧，另外我估计郑总以前就被那些劝他升级高档硬件 Power 780 的人吓怕了。非常出人意料，喜欢张嘴骂人的郑总竟然采纳了我的建议，也对我提出的让 DBA 和开发人员配合这样的要求全部满足。不过，最后他走出会议室的时候，丢下了一句狠话给我：“我现在骂人都骂累了，如果没有搞定，就走人吧。”

星期一：应用自下而上方法学，制定优化计划

立下了军令状，优化工作也就正式开始了。

下午的时候，我和小李以及来自开发商的几个开发人员在会议室进行了深入讨论，确定优化方法和实施计划。开发人员刚要开始给我详细介绍应用逻辑的时候，我立刻拒绝了，现在没有时间听这个。

其实，他们都想知道我葫芦里卖的什么药，想看看我有什么办法解决这个燃眉之急。我内心其实也是有点忐忑的，不过还是先给他们上了一课：“性能优化的方法有两种：自上而下方法学和自下而上方法学。自上而下的思路是早发现早解决，越到后面发现，优化的成本就越高，因为它是贯穿设计、开发和维护的所有阶段的，不过目前已经处于试运行阶段，自上而下显然是不可行了，所以只能采用自下而上方法学。”

我看他们似懂非懂，满脸迷茫的样子，随后接着解释自下而上方法学：“它是一种应急的办法，分别从硬件和应用入手，硬件上通过合理配置让 DB2 发挥硬件最大能力，这个不难，我用一天时间就能解决好。比较费劲的是应用优化，第一，我没有时间了解应用逻辑的细节，当务之急是需要花时间把最影响性能的模块找出来，再从这个模块里面找出前 10 位执行时间最长、执行次数最多的 SQL 语句，分析它们的访问计划，随后运用索引、表连接、分区等技术有效解决它们；第二，在开发应用的时候，开发人员对 DB2 的锁机制估计考虑的不周，上午小李提到的有 1000 并发用户时，系统响应时间急剧增加，这个很可能和锁有关，但本质上牵涉到应用代码。”

于是，大家一起制定了优化计划：星期二，调整参数，发挥硬件的处理能力；星期三，优化执行时间最长、执行次数最多的 SQL 语句；星期四和星期五，从应用角度解决锁问题。

星期二：调整参数，发挥硬件处理能力

我和小李来到了客户机房，一进到机房，让我大吃一惊。作为一名常年在客户现场服务的 DB2 工程师，我去过北上广很多大客户的机房，包括最大电信运营商的和最大银行的，这个客户在宁波，我原以为硬件投入上比不上北上广那些客户，但是没想到进去后，全是清一色的 IBM 服务器、HP 服务器，还有 EMC 存储服务器，足有 100 多台。我忍不住一声叹息，都这么多硬件了，还想忽悠客户继续升级硬件，这也太浮躁了。

言归正传，考虑到业务系统的性能不仅仅由数据库决定，它涉及存储、主机、DB2 数据库和 Weblogic 应用服务器，所以，对这些内容都要进行性能监控。

首先监控了 Weblogic 应用服务器的运行情况。主机的 CPU 和内存利用率一切正常，这个也是开发商的强项，他们对 Weblogic 非常熟悉，所以没有问题也是意料之中的事。

其次是存储规划。这块是小李搭建的，一共 16 块盘，每块盘 300GB 左右，划分为 4 个 RAID 组，每个 RAID 组是 3D+1P，其中两个 RAID 组存放数据，一个 RAID 组存放索引，最后一个 RAID 组存放事务日志，这是一种教科书式的规划。我通过 SSH 命令登录到 DB2 服务器上，使用 dd 命令对每个 RAID 组的 I/O 吞吐能力进行了测速，可以达到 200 M/s 以上，暂时没有调整的必要。

```
dd if=/dev/zero of=/data1/test.file bs=8192 count=5000000
5000000+0 records in
5000000+0 records out
40960000000 bytes (41 GB) copied, 179.411 seconds, 228 MB/s
```

接着是 CPU 利用率。在 1000 并发用户的情况下，拥有 16 个内核的 Power 740 的 CPU 利用率也就是 20%左右，显然是足够了。

最后是监控内存。使用 get snapshot 命令抓取了数据库快照和应用快照，发现缓冲池的命中率竟然只有 60%，而且有大量的排序溢出和编目缓存溢出。我仔细检查了一下，发现服务器总的物理内存为 64GB，平时可用内存大约 48GB 左右，但是发现 DB2 仅仅申请了 2G 内存，用于缓冲池、排序堆、包缓存、编目缓存、锁列表等！把这么多内存空余下来想干啥？这是最大的浪费啊，而且缓冲池命中率这么低。想想也觉得这没有什么奇怪的，很多 DBA 只知道 DB2 提供的 STMM 内存自调优，但并不了解 STMM 对并发访问量非常大的交易系统的自调整有一定的滞后，而且它本身也有一定的开销。

凭借我的技术和经验，我对下面的参数进行了手工调整。其实真正调整的就是下面几条语句，但就是这几条语句，客户、开发商和第三方顾问公司要求我写书面的调整建议书，随后反复评估了整个下午，先在测试环境验证，最后我和小李在晚上负载低的时候进行了正式实施。

调整缓冲池大小，将 DataBuf 设置为 102400 个页面，由于页面大小为 16K，所以总大小为 16GB，IndexBuf 设置为 512000，即 8GB，再次监控，缓冲池命中率达到了 99%！

```
--pagesize 为 16K
ALTER BUFFERPOOL  DataBuf IMMEDIATE size 1024000
ALTER BUFFERPOOL  IndexBuf IMMEDIATE size 512000
```

增加 sortheap 大小直到不出现排序溢出为止，最终调整为 819200。

```
update dbm cfg using SHEAPTHRES 0
update db cfg using sheapthres_shr 1638400
update db cfg using sortheap 819200
```

同样的办法，增大 CATALOGCACHE_SZ 直到编目缓存不出现溢出为止，最终调整为 102400。

```
update db cfg using CATALOGCACHE_SZ 102400
```

同样的办法，增大 PCKCACHESZ 直到包缓存不出现溢出为止，最终调整为 102400。

```
update db cfg using PCKCACHESZ 102400
```

星期三：优化前 10 位执行时间最长、执行次数最多的 SQL 语句

这一天是我过的最开心的一天，也是最顺利的一天。早上刚到机房，小李就把昨晚的对比结果发出来了：经过参数调整后，在 1000 并发用户的情况下，系统的响应时间从 4s 减少到了 2s。

接下来按照计划开始了优化 SQL 语句。本来以为能抓住什么能装满好几页的 SQL 语句，假如我自己解决不了的话，可以请教加拿大多伦多实验室的 SQL 专家，但没想到我从 snapdyn_sql 管理视图里面抓出的前 10 位执行时间最长的 SQL 语句竟然都非常简单，但是执行次数非常多，达到了上亿次！

```
    select  TOTAL_EXEC_TIME, NUM_EXECUTIONS,STMT_TEXT from sysibmadm.snapdyn_sql order
by TOTAL_EXEC_TIME desc fetch first 10 rows only
    NUM_EXECUTIONS       TOTAL_EXEC_TIME      STMT_TEXT
    3292321             293118181
    SELECT ORDERID, ORDERTIME FROM BANK.ORDER WHERE ORDERNumber = ?  AND ORDERTIME > ?
AND ORDERTIME <= ? ORDER BY ORDERTIME DESC
    ...                    ...                     ...
```

首先来看第 1 条执行时间最长的 SQL 语句。这个 SQL 语句很简单，就是对表 ORDER 上的一个动态查询，ORDER 这个表有 3 亿多条记录，它竟然执行了 3292321 秒，执行了 293118181 次！

解决办法是什么？其实很简单，为这条语句创建如下索引，这样不用再对 ORDER 表进行表扫描，可以通过索引扫描取得结果：

```
    CREATE INDEX "BANK"."IQUERY" ON "BANK"." ORDER"
             ("ORDERNumber" ASC,
              "ORDERTIME" ASC)
             MINPCTUSED 10
    ALLOW REVERSE SCANS
```

创建完毕后，运行 runstats 命令，重新收集统计信息，这样优化器在生成访问计划的时候就可以用上索引了。

其他 9 条执行时间最长的 SQL 语句，也是通过索引技术进行了优化。随后，我整理了报告发给了客户、开发商和第三方顾问公司供他们评估使用。很快，他们就做了答复，同意先在测试环境验证。完成验证后，已经夜深人静了，最后我喝着咖啡指导小李在生产库上进行了成功实施。

星期四：解决锁问题

早上来的时候，路上堵车，晚到了 20 分钟。当我刚到机房门口就听见郑总在用非常洪亮的声音给小李和开发商的开发人员讲话。他刚看到我就说："王工，这几天工作成果很显著嘛，听小李说，现在平均响应时间已经优化到 1s 左右了，看样子大功告成了，晚上请你吃宁波菜！"

看样子郑总的心情不错，但是，我知道还有优化空间，因为星期二调整参数的时候发现了大量的锁等待。凭借我的经验，锁问题的产生通常是由于表的不合理设计或者事务对表的不合理访问导致的，这才是影响并发的关键所在。我告诉郑总，革命尚未成功，宜将余勇追穷寇，

我再加把劲，看看能否再提升一下。

于是，在小李的配合下，我多次使用 db2pd 工具分析锁，发现在高并发的情况下，应用会千万次的执行同一事务逻辑，即查询热表 SALESDATA 中的某一行，随后再修改这个热表中的同样的行，这样当多个事务争抢同一行时，就会出现大量的锁等待。

通过抓取应用快照，发现这些同一事务的业务逻辑也不复杂，就是两条 SQL 语句，都是对表 SALESDATA 操作的，这个表有 8000 万左右的记录数。

```
--开始事务
业务逻辑代码...
--对表 SALESDATA 进行查询
SELECT ARRIVALATTENDED, ARRIVALFIRCODE, ARRIVALHOLIDAYSURCHARGE, NIGHTARRIVAL,
VICINITYAPPROACHCOUNTVFR, VICINITYDEPARTURECOUNTVFR, WORKSTATIONIDENTIFIER FROM
BANK.SALESDATA WHERE FLIGHTIDENTIFIER = ? AND RECORDTYPE = ?
业务逻辑代码...
--对表 SALESDATA 进行更新
UPDATE BANK.SALESDATA  SET ARRIVALATTENDED = ?, ARRIVALFIRCODE = ?,
ARRIVALHOLIDAYSURCHARGE = ?, NIGHTARRIVAL = ?, VICINITYAPPROACHCOUNTVFR = ?,
VICINITYDEPARTURECOUNTVFR = ?, WORKSTATIONIDENTIFIER = ? WHERE FLIGHTIDENTIFIER
= ? AND RECORDTYPE = ?
--结束事务
```

遇到这种情况，只能和开发人员沟通一下了。开发人员告诉我，这是业务逻辑的要求，没有办法修改代码的，所以他们建议调整一下锁有关的参数而不是修改代码。我知道这么做，只是治标不治本，不过调整也能取得一定的效果，也就答应了。

最终将 LOCKTIMEOUT 从 90 调整为 30，这里的单位是秒，90 秒的时间太长了，设置为 30 秒比较合理，这样锁等待超过 30 秒后，就会回滚事务并报锁超时，从而提升事务吞吐量。

```
update db cfg using LOCKTIMEOUT 30
```

将 LOCKLIST 调大为 40960，MAXLOCK 调大为 60，这样为锁分配更多的内存资源。

```
update db cfg using LOCKLIST 40960
update db cfg using MAXLOCKS 60
```

这个事务的频繁执行，会写大量的日志到磁盘上，分别增大了日志缓冲区（LOGBUFSZ）、日志文件大小（LOGFILSIZ）、主日志文件个数（LOGPRIMARY）和辅助日志文件个数（LOGSECOND）。

```
update db cfg using LOGBUFSZ 10240;
update db cfg using LOGFILSIZ 102400;
update db cfg using LOGPRIMARY 50;
update db cfg using LOGSECOND 30;
```

晚上，郑总开车过来，带着我、小李和几个开发人员一起去了一家著名的宁波菜馆。由于 1s 的平均响应时间已经达到了，所以大家也比较放松，吃的不错，也喝了酒。我借着酒劲，告诉郑总，越到后面，调优的成本越高，但我还要再猛攻一下，看看能否达到 0.9s 左右，至少要把锁等待消灭一批才行。

星期五：应用改造

早上来到机房，小李告诉我，昨天对锁参数的调整效果不大，但是预定的目标已经达到了，这个时候停下来客户方郑总是可以接受的，不过我可不能这样就草率放弃。

Oracle 的 DBA 一般遇到问题，第一时间想到的是 Metalink 网站，对 DB2 来说就是信息中心了。我到 DB2 信息中心网站上检索了一下，原来这两条语句是这样加锁的，在当前 CS 隔离级下，第一条 SELECT 语句根据 where 条件获取某一行上的 NS 锁，这条语句可以立即获得 NS 锁，不会出现锁等待；但是，第二条 update 语句会试图将该行上的 NS 锁升级为 X 锁，这个时候就有可能发生锁等待。

```
--开始事务
业务逻辑代码...
--对表 SALESDATA 进行查询
--这条 select 语句会根据 where 字句的条件获取指定行的 NS 锁
SELECT ARRIVALATTENDED, ARRIVALFIRCODE, ARRIVALHOLIDAYSURCHARGE, NIGHTARRIVAL,
VICINITYAPPROACHCOUNTVFR, VICINITYDEPARTURECOUNTVFR, WORKSTATIONIDENTIFIER FROM
BANK.SALESDATA WHERE FLIGHTIDENTIFIER = ? AND RECORDTYPE = ?
业务逻辑代码...
--对表 SALESDATA 进行更新
--这条 update 语句会将该行上的 NS 锁试图升级为 X 锁
UPDATE BANK.SALESDATA  SET ARRIVALATTENDED = ?, ARRIVALFIRCODE = ?,
ARRIVALHOLIDAYSURCHARGE = ?, NIGHTARRIVAL = ?, VICINITYAPPROACHCOUNTVFR = ?,
VICINITYDEPARTURECOUNTVFR = ?, WORKSTATIONIDENTIFIER = ? WHERE FLIGHTIDENTIFIER
= ? AND RECORDTYPE = ?
--结束事务
```

那么有什么变通的办法呢？根据 DB2 信息中心的提示，我做了下面的改动，第一条 SELECT 语句后面加入“with RS USE AND KEEP EXCLUSIVE LOCKS”子句，这样可以直接获取指定行上的 X 锁，当然这个时候也会出现一定数量的锁等待；等到 update 语句执行的时候，无需从 NS 锁到 X 锁的转换了，也就不会再出现锁等待了。

```
--开始事务
业务逻辑代码...
--对表 SALESDATA 进行查询
--这条 select 语句会根据 where 字句的条件直接获取指定行的 X 锁
SELECT ARRIVALATTENDED, ARRIVALFIRCODE, ARRIVALHOLIDAYSURCHARGE, NIGHTARRIVAL,
VICINITYAPPROACHCOUNTVFR, VICINITYDEPARTURECOUNTVFR, WORKSTATIONIDENTIFIER FROM
BANK.SALESDATA WHERE FLIGHTIDENTIFIER = ? AND RECORDTYPE = ? with RS USE AND KEEP
EXCLUSIVE LOCKS
业务逻辑代码...
--对表 SALESDATA 进行更新
--该行上已经是 X 锁了，不需要从 NS 到 X 的锁转换了
UPDATE BANK.SALESDATA  SET ARRIVALATTENDED = ?, ARRIVALFIRCODE = ?,
ARRIVALHOLIDAYSURCHARGE = ?, NIGHTARRIVAL = ?, VICINITYAPPROACHCOUNTVFR = ?,
VICINITYDEPARTURECOUNTVFR = ?, WORKSTATIONIDENTIFIER = ? WHERE FLIGHTIDENTIFIER
= ? AND RECORDTYPE = ?
--结束事务
```

新的方案是否对性能有帮助，只有经过测试才知道。上午的时候，我把报告发给了客户和

开发商供他们评估。尽管改动很小，由于目标已经达到，所以开发商的态度还是十分消极的，不太愿意修改应用代码。不过还好，客户方的郑总尽管不懂技术细节，但他认可这个办法，至少是可以试试的。

开发商花了 30 分钟就改好了代码，重新部署到 Weblogic 应用服务器上，随后在测试环境进行了验证，取得了不错的效果。最终，星期五的晚上，我、小李和开发人员一起在生产库上进行了正式实施。

实施完毕后，已经晚上 8 点多了，我踏上了从宁波飞往北京的最后一班飞机，心中安慰自己说，至少效果不会更坏吧。到了星期天，小李打电话告诉我效果：锁等待数量从原来的几十万个减少到了几千个，平均响应时间达到了 0.9s!

DBA 要失业了吗？

本书介绍的 DB2 设计与性能优化方法，基本上都需要 DBA 调整系统参数，调整逻辑设计和物理设计，解决锁问题，定位 SQL 语句，分析其访问计划，在测试环境验证，在生产环境实施，最后评估结果。这是一种手工优化方式，对 DBA 和开发人员有一定的技术要求。

从 V9.1 开始，DB2 提供了很多自调整特性，例如自动存储管理（ Automatic Storage）、自动内存调优（Sefl Tuning Memory Management）、自适应压缩等，另外还提供了自动监控工具 OPM（Optim Performance Manager），自动 SQL 语句调优工具（Optim Query Tuner）、自动查询负载调优工具（Optim Query Workload Tuner），也就是说要从传统的手工优化时代进入自动优化时代！

本人在 2010 年参与过一个电信行业的数据仓库项目，当时客户方的 DBA 要求在项目中使用图形化工具。一名来自开发商的 DB2 高手站起来大声抗议："图形化工具没用，DB2 中的每条命令我都是顺手敲来！"大家都知道，他的确是技术精湛的高手，对他来说敲命令是没有任何难度的。当然，他说的也未必全是错的，他可以在一些对安全管理非常规范的客户那发挥关键作用。例如金融证券业客户，一般不允许你使用自己的笔记本，你只能使用客户提供的客户端软件。但是这个客户端可能什么工具都没有给你装，你只能在上面敲入字符。这种情况下，你就必须放弃图形界面这个拐杖了，离开拐杖，你也要会走路啊。

但是，对于刚接触 DB2 的新手来说，全部敲命令是不现实的，而且 DB2 的功能越来越多，在使用上只靠敲命令很难保证工作效率和项目进度的要求。所以，在为允许使用图形化工具的客户服务时，可以尽量使用工具以提高工作效率。

言归正传，那么从手工优化进入到自动优化时代，DBA 真的要失业了吗？我认为，在自动优化时代，DB2 提供的自调整特性以及一些 SQL 自动优化工具仍然无法代替设计人员、开发人员和 DBA 更富有创造力的工作。DB2 提供的自调整工具是需要人去运用的，SQL 语句的优化和业务逻辑是强关联的，是需要和设计开发人员认真沟通的，自动化工具不可能知道广大设计开发人员如何考虑的。

笔者认为，在自动优化时代，DBA 的作用将更加重要。这些自动化特性或者工具的目的只是帮助 DBA 从琐碎的事务性工作中解脱出来，这样 DBA 可以集中精力放在数据库架构、数据模型设计、应用开发指导、运维规范、优化指南等更需要智慧的地方。

本书内容

本书内容严谨精细、生动活泼，从内容来看，共分为四大部分，11 章。第一部分包括第 1 章和第 2 章，提出了两种性能优化方法学，包括理想化的自上而下方法学和救急专用的自下而上方法学，随后通过一个真实的实战案例，阐述了如何灵活运用方法学。第二部分是设计部分，包括第 3 章~第 5 章，分别谈到了物理设计、基础逻辑设计和高级逻辑设计，这是设计一个高质量的数据库系统所必须掌握的内容。第三部分是性能优化部分，包括第 6 章~第 9 章，讲述了如何对 DB2 进行性能监控，如何调整参数和优化维护工具，如何调整锁和日志来优化高并发系统，如何来优化最耗资源的 SQL 语句。第四部分是有关高级设计与优化内容，包括第 10 章和第 11 章，讲述了如何设计和优化大容量数据仓库，讲述了如何设计和优化 DB2 pureScale 集群。

为了方便读者，我们在书中加入了“注意”、“小技巧”和“比较”这样的提示性文字。其中“注意”的内容是需要 DBA 在工作中重点留意的；“小技巧”的内容是我们重点向读者分享的实战技巧；“比较”的内容是一些容易混淆的概念，是向广大 DBA 和开发人员解释和澄清的。

最后，我们在文中的一些脚本片段，采用灰底形式，这样就非常明显的和正文区别开了，同时加黑处理了脚本中的一些重点内容以示强调。

本书读者

本书目标读者群主要针对数据库管理员和应用设计开发人员：

（1）数据库管理员：中国的数据库管理员非常勤奋，非常能吃苦，对 AIX、Linux 等操作系统掌握的很不错，对 DB2 的安装配置也都没问题，这是他们的强项。但是到了运维阶段遇到性能问题就只能苦苦摸索了，完全靠“野路子”打游击战：从网上搜点没有经过验证的性能优化秘籍，就敢在生产系统上直接实施，成功了是运气好，失败了算自己倒霉。另外，我也见过很多数据库管理员，不了解应用的设计和开发，丝毫不关注 SQL 语句背后的业务逻辑，这说明还缺乏自上而下方法学的熏陶。因此，这本 IBM 原厂的性能优化专著将会帮助您们掌握原厂“正规军”的战略战术。

（2）设计开发人员：目前设计开发人员的主力都是 80 后和 90 后了，尽管非常年轻，他们中的很多人已经开始承担核心应用的开发了，我非常理解由于用户需求的频繁变更，让开发人员苦不堪言，所以没有时间或者不屑去关注 SQL 语句的数据库运行环境，可能认为只要掌握好 Select、Insert、Update 和 Delete 这 4 条 SQL 语句就可以打遍天下无敌手了，但是我想说的是，DB2 是 IBM 公司数以千计的工程师历经几十年打造的关系数据库精品，里面的技术含量很高，不是为了仅仅支持那 4 条 SQL 语句的！我建议设计开发人员在繁忙的设计开发工作之余，阅读一下本书，这本书将会从应用的角度帮助您真正掌握 DB2 设计与性能优化技术，读后一定会豁然开朗，收益颇丰。

致谢

感谢我在 IBM 公司的同事侯战友，很多性能优化思路我都和他讨论过，他的真知灼见让我受益匪浅。本书侧重于 DB2 设计和性能优化，但也参考过《运筹帷幄 DB2》一书中的有关 DB2

运维内容，这里我想特别感谢我在 IBM 公司的同事孙岳，本书参考了他在备份恢复以及日志机制方面的一些技术写作成果；感谢我在 IBM 公司的同事刘旭，本书参考了他在数据移动方面的一些技术写作成果；感谢我在 IBM 公司的同事张广舟，他在 DB2 性能监控和锁并发机制方面具有非常深入的造诣，本书参考了他在这方面的一些技术写作成果。我还要感谢我在 IBM 公司的同事杨小华、孔再华和毛青，他们是 DB2 pureScale 方面的专家，我向他们学习了很多 DB2 pureScale 方面的技术。另外，我还要感谢 IBM 公司的存储专家张晓巍，他在存储设计上给予了我们大力指导。最后，我还要感谢 itpub、db2china、中国 DB2 技术交流群（QQ 群，186465431）等社区上的朋友们，我从和他们的技术探讨中获得了很多写作灵感。

王飞鹏 2013 年 7 月写于北京

目 录

第 1 章 性能优化方法学

第 2 章　实战案例研究与分享

第 3 章　高质量物理设计

第 4 章 经典逻辑设计

第 5 章 高级逻辑设计

第 6 章　系统监控

第 7 章 配置参数与运维工具优化

第 8 章 锁和日志优化

第 9 章 SQL 语句优化实战

第 10 章 DB2 数据仓库设计与优化

第 11 章 DB2 pureScale 集群数据库

后记　信念的奇迹

缩略语

参考文献

第1章 性能优化方法学

我们经常遇到这样的情况：数据库应用系统上线运行一段时间后，就会出现性能问题，例如处理相同的查询比刚上线要长很多，在业务繁忙时事务吞吐量无法满足实际需要，有时候严重的性能问题甚至导致数据库宕机。往往在这个时候，客户才意识到，在设计开发阶段，数据库系统的性能需求没有得到应有的重视。

当出现这样的性能问题时，有的 DBA 在慌乱之中对数据库进行盲目的参数调整，这可能让情况更加糟糕；也有的 DBA，要求将运行数据库的现有硬件替换成高档硬件，这很可能是一种资源的浪费，因为有时候即使换了高档硬件，性能问题还是依旧存在。

归根结底，他们急需一种满足实战要求的性能优化方法学。

1.1 什么是性能问题

从大的方面来看，性能问题往往分为两大类：影响了整个系统的问题和影响了部分应用的问题。影响了整个系统的问题，例如整个系统的 I/O 瓶颈；影响了部分应用的情况，例如某一特定应用或 SQL 语句。

性能问题比功能性问题难解决，最首要的原因在于，性能问题往往是断断续续发生的，不容易监控和捕捉。其次，根据症状不容易找到问题的真正根源。例如有时候发现数据库系统有大量的锁超时症状，真正根源却不是锁的问题，而是由于过时的统计信息导致优化器选择了非最优的访问计划导致的，这对 DBA 来说往往是很难定位的。

本节将首先探讨如何发现系统的性能问题，随后讲述衡量性能的关键指标，如何进行性能基准测试，最后探讨如何制定优化目标。

1.1.1 如何发现性能问题

性能问题的背后实际上是业务驱动的。所以，可以从业务角度确定问题，设定每个问题的优化目标，设定每个问题的优先级，随后从系统角度确定原因，弄清楚时间花在何处；资源是如何被消耗的。

具体来说，可以先从下面的问题入手，来判断系统是否存在性能问题。

（1）是否有性能降低，与什么相关？“基准”是什么？

（2）一个系统的性能看起来在随时间流逝下降了？还是不同的系统性能比较，发现有一个系统性能下降了？

（3）性能下降是什么时候发生的？与某个工作负载的运行相关吗？性能下降周期性地发生吗？

（4）性能下降的前后系统有什么变化吗？例如添加了新硬件、或应用程序被更改了、大量数据被加载、或者更多的用户访问这个系统？

（5）能不能向数据库管理员、应用程序开发人员以及架构师多了解一些业务和系统情况？因为 DB2 服务器几乎总是硬件、中间件和应用程序这样复杂环境的一部分。

1.1.2 衡量性能的关键指标

一个系统的资源例如存储、CPU 和内存等越多，资源被调度的越有效，能处理的负载就越大。换句话说，性能受两个关键因素的影响。其一是系统可用资源，其二是这些资源是如何被使用和共享的。

从量化指标来看，性能的好坏可以从以下三方面来衡量：

（1）响应时间：应用向数据库服务器发起请求，服务器完成处理并返回结果给应用，中间总共消耗的时间，它反映了数据库服务器的处理速度。

（2）事务吞吐量：通常用每分钟处理的事务数（TPM）或者每秒钟处理的事务数（TPS）来计算，这个指标反映了系统的事务处理能力。

（3）资源利用率：数据库服务器在处理事务或者查询的过程中，系统资源包括 CPU、I/O

以及磁盘的使用情况，这个指标反映了系统资源是否被有效利用。

1.1.3 性能基准测试

制定性能优化目标之前，首先需要对系统做基准测试（Benchmark Test，即 BMT）。基准测试是一种测量和评估软件性能指标的活动，它通过科学的测试方法、测试工具和测试系统，实现对某项性能指标进行定量和可对比的测试。

基准测试能帮助用户理解数据库在不同条件下的性能情况。例如通过基准测试建立一个已知的性能水平（称为基准线），当系统的软硬件环境发生变化之后再进行一次基准测试以确定那些变化对性能的影响，这是基准测试最常见的用途。

基准测试的时机很多，可能是数据库配置参数更改，也可以是应用部署发现变化，或者是数据版本升级了。这个时候可以将再次测试的性能值和变化前进行对比，并做定量分析以确认性能变化情况。例如在开发阶段，使用基准测试来确定应用程序是否出现性能倒退；在运维阶段，使用基准测试来对调优前后的结果进行对比。

> **注意：基准测试不仅仅是 DBA 的工作**
>
> 基准测试是数据库应用生命周期的一部分，不仅需要 DBA 的参与，也需要设计人员和应用开发人员的积极参与和配合。

1.1.4 制定优化目标

制定性能优化目标的关键在于量化。不论是整个系统的问题，还是只影响了部分应用的问题，首先都要确定响应时间能比基准值减少多少，其次要确定事务吞吐量能比基准值提升多少。

根据系统性能的基准值以及系统现状，就可以灵活制定性能优化目标了，比如：

（1）现有系统性或者部分应用存在性能问题，制定优化目标。例如要求系统优化后，响应时间要比基准值减少 20%，事务吞吐量则要在基准值的基础上增加 20%以上。

（2）现有系统或者部分应用发生大的变更后，制定优化目标。例如新上线了一个模块，系统的负载变重，但是硬件资源没有发生变化，这时的性能目标可以定义为：新模块上线后响应时间要和基准值相似，可接受的幅度在 5%左右，但是总的事务吞吐量要求在基准值的基础上增加 20%。

（3）对现有系统或者部分应用进行优化，到了一定阶段后，制定后续目标。例如，目前性能已经达到基准值的 150%了，考虑到边际效应，以后的优化只能产生越来越少的效益。优化无止境，当 DBA 考虑需要多少时间和金钱用于性能优化时，要评估一下所花费的时间成本和金钱成本。

1.2 性能优化方法学

我和很多 DBA 交流过，有些人认为性能优化很高深，需要了解 DB2 的配置参数、锁、交易日志等技术细节，通常持有这种观点的人都是数据库高手，而且他们也以掌握了这些细节而

自豪。

真实情况是，性能优化没有那么神秘，未必一定要掌握那么深入的技术细节。大家知道，性能问题的解决是找到瓶颈和解决瓶颈的过程，只要在应用层和数据库层找到了真正的瓶颈，随后使用正确的方法学加以优化，不用完全掌握那些技术细节，也能有效地解决问题。

本节首先谈谈几种常见的性能优化误区，随后探讨一下两种性能优化方法学，即自上而下方法学和自下而上方法学，最后阐述如何在性能优化时灵活选择它们。

1.2.1 几种常见误区

如果没有掌握正确的方法学，在实际生产环境中定位和解决性能问题不是一件容易的事。我们接触了很多客户，在性能优化时存在下面三种典型的误区。

（1）性能优化可以通过参数调整来解决所有问题

客户通常都热切希望不要动他们的硬件和应用，DB2 有什么内部参数调整一下，就可以解决所有的性能问题。这种观点是不正确的，数据库参数要调整，但即使调整好了也并不能保证解决所有的性能问题。

（2）性能优化是 DBA 的工作，不需要设计人员和开发人员参与

这个观点认为，在项目进行当中，设计人员只关注数据模型，不用关心逻辑设计和物理设计；开发人员只关注业务功能，不用关心 SQL 语句的性能问题。但是，这种观点是十分错误的，大多数系统的性能问题还是由于设计和应用导致的，所以在优化工作中，需要设计人员和开发人员的积极参与。

（3）用更快的硬件可以解决所有的性能问题

持有这种观点的人很多，他们认为出现性能问题后，用更快的 CPU、更多的内存和更快的 I/O 设备就能解决。但是，这种说法是不全面的，例如对于锁问题造成的懒惰系统，增加硬件资源是不能解决的。实际上，需要找到性能问题的真正原因，再采取恰当的措施。

注意：什么时候选择硬件扩容？

如果确认系统在应用或者数据库层面已经没有优化空间了，这时可以考虑硬件扩容的方式。例如针对性地增加存储，使用更快的 CPU、更多的内存、更快的网络连接，或者上面三者的组合。

1.2.2 自上而下（From Top To Down）方法学

一个系统进入正式运维阶段前，通常会经历设计、开发和上线等阶段，自上而下方法学是一种理想的方法论，要求在各个阶段不仅要考虑业务功能的实现，而且要同时考虑性能。例如，在设计阶段考虑数据库逻辑设计、数据库物理设计、应用设计等；在开发阶段，考虑如何编写高效的 SQL 语句，如何减少死锁等；在上线阶段，考虑内存和 CPU 的优化，考虑磁盘 I/O 设计和调整等。

下面几点也是自上而下方法学重点强调的：

（1）优化工作一定要贯彻始终，越早越好，因为开始的越早，其付出的成本也就越小。

（2）系统上线前，应尽可能的解决好所有的性能问题，否则上线后出现问题，往往是灾难性的。

（3）设计和开发人员需要认真对待设计和开发阶段的性能问题，不能完全依靠性能更强的硬件解决。

1.2.3　自下而上（From Down To Top）方法学

与自上而下不同，自下而上方法学是从硬件和操作系统开始，接着是数据库，随后是应用服务器，再到应用这样的顺序去诊断问题，根本目标是找到性能瓶颈。很多 DBA 都有这样的感觉，当出现性能问题时，找设计和开发人员帮助，有时候很难，因为这些人在正式上线后大部分已经退出项目组了，但是找系统管理员和网络管理员帮助是非常容易的，在大部分单位都可以做到。所以，这种方法学对 DBA 来说容易上手，其难点在于找性能瓶颈的过程。

该方法学在解决性能问题时，共分为四步：

（1）第一步，问题监控：弄清楚系统的整体运行情况，包括硬件、操作系统、数据库、中间件和业务，确定性能瓶颈到底在哪一个层面。

（2）第二步，参数调整：找到性能瓶颈后，如果发现可以通过参数调整来解决，那么优先通过参数调整来低成本地解决问题，通常这些参数包括硬件参数、操作系统参数、DB2 参数、中间件参数等。

（3）第三步，数据库优化：如果性能瓶颈在数据库层面，并且参数调整无法解决问题，这时就要根据具体的瓶颈种类进行灵活处理，具体的处理办法见本章 1.4 节。

（4）第四步，性能验证：优化完毕后，将优化指标和优化前的基准值进行对比，没有必要片面追求最高性能，只要当前性能指标满足应用的真正需要即可。

1.2.4　自上而下和自下而上，如何选择？

从技术上来看，这两个方法学的目的都是为了解决性能问题，只是顺序不同，一个是自上而下，一个是自下而上。自上而下是理想的方法，自下而上通常是不得已而为之。如果多用自上而下，需要救火的情况就会少，否则不可避免地要会采用自下而上的方法来救火。不过，在具体的项目中，技术角度只是考虑的一个层面，往往要根据实际情况进行选择和取舍。

（1）如果是一个新系统，那么可以采用自上而下的方式，在设计、开发和运行维护的各个阶段不仅考虑功能实现，也要考虑性能问题。

（2）如果是系统已经上线并且运行一段时间了，性能问题变得严重后才被关注，这个时候其实就是一个现场救火了，采用自上而下几乎是不可能的，客户没有时间或者也不可能把设计、开发和上线情况说清楚，所以只能采用自下而上方法学。

（3）有时候，当优化人员参与一个迭代式项目的时候，不同模块处于不同的生命周期。例如有的模块也许刚进入设计和开发阶段，这个时候适合采用自上而下方法学；但是有的模块已经过了设计和开发阶段了，进入上线试运行阶段了，这个时候只能采用自下而上方法学。

1.3　高质量数据库设计

如果要建设一个新的数据库系统，那么必然会经历设计、开发和维护的各个阶段，按照自

上而下方法学，在考虑业务功能实现的同时，性能优化应贯彻始终，越早越好，因为开始的越早，其付出的成本也就越小。

本小节将按照自上而下方法学的实践方法，深入探讨一下如何进行高质量的数据库设计。

注意：自上而下方法学施行起来不容易

自上而下方法学是性能优化理论上最好的方法，它要求在设计、开发和维护的各个阶段都要重视性能问题，在这种方法学指导下实施的数据库通常性能较高，而且在运维阶段出现性能问题的概率要小很多。

但是，这种方法学施行起来不容易，它需要设计人员、开发人员和 DBA 的通力协作。现实中，设计人员往往只关注数据模型；开发人员往往只关注业务功能的实现；到了上线运行后，只能靠 DBA 来四处救火了。

先看一下数据库的设计过程，如图 1-1 所示。首先需要以用户的眼光来看待现实世界的业务，做到真正理解客户的业务需求，这就是通常所说的收集需求阶段；随后业务人员根据业务需求设计概念模型，并且该模型需要和现实世界相对应，这就是概念模型设计阶段；接着进入逻辑结构设计阶段，在这个阶段数据库设计人员使用业务人员的概念设计结果，将其转换为具体数据库的对象结构，例如表、视图等；然后是物理结构设计阶段，需要为逻辑模型选取一个最适合的物理环境。

设计完成后，就进入开发阶段了，这个阶段 DBA 会将前面物理设计的结果部署到目标机器上，开发人员开始编写应用代码去实现业务逻辑，经过测试后准备上线；最后是运行维护阶段，这个阶段主要是上线后的日常监控、维护和优化工作。

本节将从自上而下优化方法学的思路出发，谈谈在各个阶段需要特别注意的重要事项。

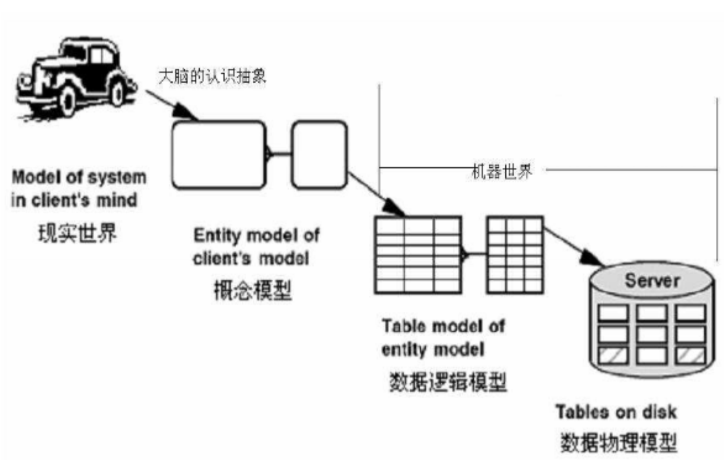


图 1-1 数据库设计的一系列阶段

1.3.1 充分了解需求

这个阶段的目标是准确了解用户的需求，它是整个数据库设计过程的基础，也是最困难、最耗费时间的第一步。这个阶段主要由业务人员完成。首先调查组织机构情况、各部门的业务活动情况、协助用户明确对新系统的各种要求、确定新系统的边界。随后调查、收集与分析用户在数据管理中的信息要求、处理要求、安全性以及完整性要求，最终得到数据流图表达的数据和处理过程的关系。

除了上述功能需求外，性能需求也需要在这个阶段被定义。通常包括并发要求、响应时间要

求、数据库容量要求、I/O 吞吐能力要求、系统用户容量要求和系统运行时间要求，例如 7×24 小时不间断运行，或者可连续运行一周或一月。

在现实的设计实践中，如何才能获得真正有效的性能需求呢？这需要从以下三个方面加以考虑。

（1）性能指标必须量化

指标量化应该不难理解，但是有了数字并不代表就实现了真正的量化。例如常见的一种需求是“系统需要支持 5000 用户”，或者“最大在线用户数为 8000”。这些有数字的需求仍然不够明确，因为还需要区分在线用户和并发用户的区别。为便于理解，先来看一个性能需求的例子。

某个金融行业交易系统的需求：

① 系统总容量达到日委托 9000 万笔，成交 9000 万笔。

② 系统处理速度每秒 7300 笔，峰值处理能力达到每秒 10000 笔。

③ 实际股东帐号数 3000 万。

这个例子中已经包括三个明确的性能需求：

① 交易吞吐量：日委托和日成交都是 9000 万笔。

② 平时处理需求：每秒 7300 笔；忙时处理需求：峰值达到每秒 10000 笔。

③ 用户容量：实际股东帐号数 3000 万。

（2）有实际操作意义

一般来说，性能需求要么由集成商提出，要么由客户提出。很多集成商由于项目进度的压力，往往对性能需求的定义不太严格，靠拍脑袋做决定，这是错误的，一定要确保在定义性能需求时，所使用的数据和计算公式是有根据的。对于由客户提出的情况，例如电信、银行、保险、证券以及一些其他运营商级系统的客户，他们和集成商不一样，他们往往对性能需求非常认真，但是这又是另外一个极端，经常提出超过实际需要的性能需求，这个时候，需要耐心做好说服工作，让客户明白性能是有成本的，没必要追求超过实际需要的性能。

（3）对性能需求的理解具有一致性

相关人员对性能需求的理解不一致通常是由于参考对象的不同导致的。例如有的人员根据客户以往的业务情况来分析客户的业务量；有的人员参考其他规模类似客户的情况；有的人员参考其他同行企业公布出来的数据；有的人员参考类似行业的应用。这就需要相关人员使用相同的参考对象来定义性能需求，否则很难达成一致。

1.3.2　设计概念模型

概念模型是按用户的观点来对数据和信息建模，通常用实体-关系图（E-R 图）表示。通过对用户需求进行综合、归纳与抽象，形成一个不依赖于具体数据库管理系统（DBMS）的概念数据模型，但可以转换为某一 DBMS 支持的特定数据模型。概念模型具有较强的语义表达能力，能够方便、直接地表达应用中的各种语义知识。除此之外，概念模型清晰、易于用户理解，是用户与业务人员之间进行交流的语言。

1. 定义实体

实体都有一个共同的特征和属性集，可以从需求收集阶段收集的基本数据资料表中直接或

间接标识出大部分实体。例如客户代码、代理商代码、产品代码等将其名词部分代表的实体标识出来，从而找出潜在的实体，形成实体表。

2. 定义关系

关系模型中只允许二元关系，n 元关系必须定义为 n 个二元关系。根据实际的业务需求和规则，使用实体联系矩阵来标识实体间的二元关系，然后根据实际情况确定出关系名和说明。这里的关系类型需要明确，是标识关系还是非标识关系？如果是非标识关系，是强制的还是非强制？如果子实体的每个实例都需要通过和父实体的关系来标识，则为标识关系，否则为非标识关系。非标识关系中，如果每个子实体的实例都与而且只与一个父实体关联，则为强制的，否则为非强制的。

3. 定义码

通过引入交叉实体除去上一阶段产生的非确定关系，然后从非交叉实体和独立实体开始标识候选码属性，以便唯一识别每个实体的实例，再从候选码中确定主码。为了确定主码和关系的有效性，通过非空规则和非多值规则来保证，即一个实体实例的一个属性不能是空值，也不能在同一时刻有一个以上的值。

4. 定义属性

从基本数据表中抽取说明性的名词开发出属性表，确定属性的所有者。定义非主码属性，检查属性的非空及非多值规则。此外，还要检查完全依赖函数规则和非传递依赖规则，保证一个非主码属性必须依赖于主码。以此得到了至少符合关系理论第三范式的改进的全属性视图。

5. 定义其他对象和规则

定义属性的数据类型、长度、精度、非空、缺省值、约束规则等。

1.3.3 设计逻辑结构

本书第 4 章和第 5 章将深入探讨如何设计高质量的逻辑结构。简而言之，设计逻辑结构首先选择最适合描述相应概念结构的数据模型。数据模型是数据库系统的核心，主要包括网状模型、层次模型、关系模型等，DB2 就是基于关系型数据模型实现的。

数据模型优化的关键在于确定数据依赖，消除冗余的联系，利用范式对数据模型进行规范化。数据库应用的类型不同，所采用的范式也会不同。

数据库系统从应用上分为事务型的（On-line Transaction Processing，简写为 OLTP）和分析型的（On-line Analysis Processing，简写为 OLAP）。

由于 OLAP 应用和 OLTP 应用有本质差异，所以在逻辑设计上也不同。OLTP 应用通常需要尽量减少冗余，所以采用第三范式；OLAP 应用中的表结构采用星型或雪花型模型，以方便查询和多维分析，例如业务数据（例如成本、收入）存储在事实表中，而这些事实数据所依赖的属性（例如时间、地区、产品、客户）存储在维表中。

比较：OLTP 应用和 OLAP 应用的区别

OLTP 应用的用户是业务人员，要求非常快地处理查询和事务，ERP 和 CRM 系统是典型的 OLTP 系统；OLAP 应用的用户是决策管理人员，要求处理复杂查询以及报表等。关于它们之间的区别如表 1-1 所示。

表 1-1 OLTP 和 OLAP 应用之间的主要区别

特　　点	OLTP 应用	OLAP 应用
负载特点	日常业务操作	查询、分析、报表和批量数据加载
SQL 语句种类	SQL 语句比较简单，主要是插入和修改操作	SQL 语句比较复杂，主要是查询操作
事务吞吐量	非常高	不高
单笔事务的资源利用率	小	大
索引	少量索引	索引比较多
数据规范化	第三范式	星型或雪花模型
分区技术	表分区使用较多	DPF、表分区和 MDC 结合使用
物化视图	很少	很多

逻辑结构的设计过程就是将 E-R 图转换为关系模型，即将实体、实体的属性和实体之间的联系进行转化。这种转换一般遵循如下原则：一个实体转换为一个关系模式，实体的属性就是关系的属性，实体的码就是关系的键值。最后，这些关系模型体现在对具体 DBMS 的表设计上。逻辑设计的过程可分为下面几个步骤：

（1）表设计

在关系表中，表中的每个数据行都是相关数据值的集合。一个表的每一列都必须具有对于该表唯一的名称。数据类型和长度指定对该列有效的数据类型和最大长度。主键值是唯一的，一个表上只能有一个主键。

（2）表之间的关系

主要包括一对一、一对多和多对多。一对一关系在两个方向都是单值的，例如一个经理管理一个部门，一个部门只有一个经理。一对多关系中，一个职员只能在一个部门工作，对于职员，此关系是单值的；另一方面，一个部门可有许多职员，对于部门，此关系是多值的。两个方向都是多值的关系是多对多关系。一个职员可以处理多个项目，而一个项目可以有多个职员。

（3）数据完整性设计

约束使 DBMS 能够防止不正确的或意外的数据输入表中，从而确保表中数据的完整性。DB2 有下面几种类型的约束。

- 唯一约束（unique constraint）防止一个值在表中的特定列里出现不止一次。必须将唯一约束中所引用的列定义为非空（NOT NULL）。
- 引用约束（referential constraint）用来在表之间建立引用关系，这两个表通常称为子表和父表。当在父表或子表中插入/删除/更新的数据满足预定义的条件时，就执行这

个操作。

- 表检查约束（table check constraint）针对一个或多个表列定义检查约束，它们可以对列实施指定的规则，使这些列中插入或更新的数据满足检查约束中预先定义的条件。
- 信息约束（informational constraint）是 SQL 编译器的规则，依赖于具体的 DBMS 厂商的实现，通常不需要设计开发人员干预。

（4）索引设计

针对数据的查询和应用需要，在表的某些列上建立索引或组合索引。

（5）数据组织方案

DB2 提供了一种三级数据组织方案，这些组织方案单独使用，也可以同时使用。如下所示：

- 数据库分区（DPF）：数据均匀地分布在数据库分区上，以启用查询内并行性并平衡每个数据库分区上的负载。
- 表分区（Table Partition）：同一个表中的数据可以根据键值的范围放到不同的表分区中。
- 多维集群（MDC）：将多个维上具有相同值的行放到同一个物理块中。

1.3.4 设计物理结构

逻辑数据模型是数据的理想蓝图，物理数据模型才是对数据的物理实现。物理结构设计中需要重点考虑的问题，本书第 3 章将进行深入探讨。

在数据库系统建设初期，系统架构师需要根据当前的业务情况、对未来发展的规划以及建设经费等情况进行评估。在硬件选型时，不能盲目选择高档硬件，要根据评估结果科学选择，具体硬件选型请参考本书第 3 章的有关内容。

完成硬件选型后，接下来就是根据 DB2 的特点和处理的需要，进行存储规划、缓冲池规划和表空间规划等。

（1）存储规划：对于数据库系统来说，存储规划至关重要，首先要设计合理的 RAID 级别，这样可以实现磁盘阵列的高效读写。随后在磁盘阵列上规划好这几个重要的存储空间，包括实例目录、数据库目录、数据、活动日志、归档日志、备份以及临时数据等。

（2）内存设置和规划：主要完成实例共享内存、数据库共享内存、应用程序全局内存和代理私有内存的设置，除此以外，要规划好缓冲池，它对性能影响最大。当然完全靠 DBA 来设置内存比较复杂，所以可以结合 DB2 提供的自调优内存管理（STMM），这样可以大大提高效率。

（3）表空间规划：DB2 中有两种类型的表空间，分别是系统管理的表空间（SMS）和数据库管理的表空间（DMS）。它们之间的区别在于，DMS 表空间所需要的空间是在创建时分配的；SMS 表空间是根据需要分配的。在创建表空间时，需要指定存放容器，这里容器可以是目录名、设备名或者文件名。为了提高性能和更灵活地配置，单个表空间可跨多个容器。由于 DMS 表空间性能更高，所以大部分情况下使用 DMS 表空间是一种最佳选择。

1.3.5 应用开发阶段

设计完成后，就进入开发阶段了，这个阶段 DBA 会将物理设计的结果部署到目标开发机

器上，随后开发人员开始编写应用代码以实现业务逻辑。

很多开发人员都遇到过这样的情况：在应用系统开发初期，由于开发数据库中的数据比较少，对于查询语句和存储过程等的编写体会不出 SQL 语句各种写法的性能优劣，但是如果将应用系统提交上线使用后，随着数据库中数据的增加，SQL 语句的优劣很容易就体现出来了。特别是面对海量数据，劣质 SQL 语句和优质 SQL 语句之间的速度差别可以达到上百倍，可见对于一个系统不是简单地能实现其功能即可，而是要写出高质量的 SQL 语句，提高系统的可用性。

所以，在这个阶段，不仅要做好功能测试，还要模拟生产环境的数据量做好集成测试和性能测试，直到所有测试完成后才能正式上线。

1.3.6　运行维护阶段

这个阶段主要是完成数据库和应用的部署工作，随后上线试运行，通过试运行来验证设计和开发是否满足了需求，并做必要的调整。经过试运行后，数据库应用系统即可投入正式运行。在数据库系统运行过程中还需要持续对系统以及工作负载进行性能监控、诊断与优化。

1.4　性能调整与优化

如果数据库系统已经处于维护阶段了，这时遇到了性能问题，使用自上而下方法学肯定是来不及了，只能采用自下而上方法学，使用这种方法学，其目标是找到性能瓶颈并加以解决。

如图 1-2 所示，生产环境中遇到的性能瓶颈可以归结为下面五种类型：

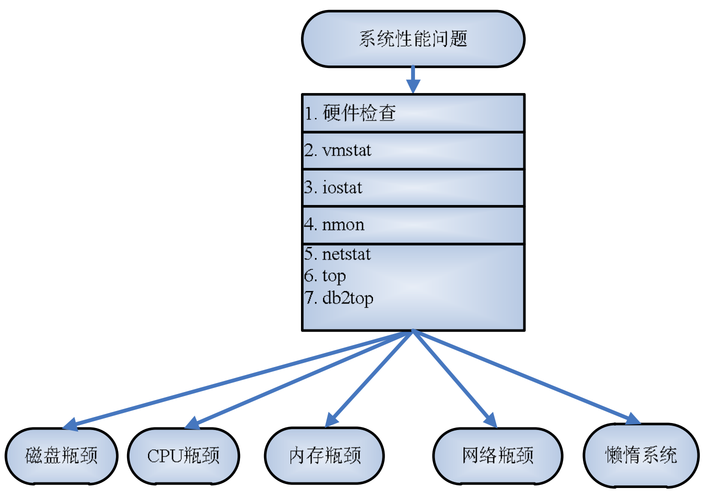


图 1-2　性能瓶颈种类

（1）磁盘瓶颈：磁盘 I/O 是数据库性能的瓶颈。

（2）CPU 瓶颈：CPU 是数据库性能的瓶颈。

（3）内存瓶颈：内存是数据库性能的瓶颈。

（4）网络瓶颈：客户端和数据库服务器之间，或者 DB2 pureScale 集群各个成员和 CF 之间，或者数据仓库 DPF 节点之间存在网络瓶颈，就会导致数据库性能变慢。

（5）“懒惰系统”：数据库运行变慢，但从监控来看，磁盘、CPU 和内存都很空闲。

1.4.1 磁盘瓶颈

磁盘瓶颈的基本症状是：在 vmstat 或 iostat 结果中，出现较长的 I/O 等待时间，较低的 CPU 利用率（25%~50%），磁盘较高的繁忙程度(80%)。对于磁盘瓶颈，首先需要找到繁忙设备所在的文件系统路径，接下来，查看在这些文件系统路径上都驻留了哪些数据库对象，例如热表、热索引、临时表、日志。最后，根据这些数据库对象的不同进行针对性的分析。

1. 热表导致的瓶颈

检查繁忙设备所在的文件路径上都有哪些数据表空间，这些表空间上都有哪些热表，找到热表后进一步向下钻取，找出什么造成了这些表的高度活跃。是动态 SQL 语句造成的高度活跃？还是静态 SQL 语句导致的高度活跃？一旦确定一个或多个 SQL 语句导致了 I/O 瓶颈，下一步需要确认这个语句是否可以被优化以降低 I/O。例如，这个语句是否发起了一个针对大表的表扫描？是否可以通过增加缓冲池大小来消除 I/O 瓶颈？整个过程如图 1-3 所示。

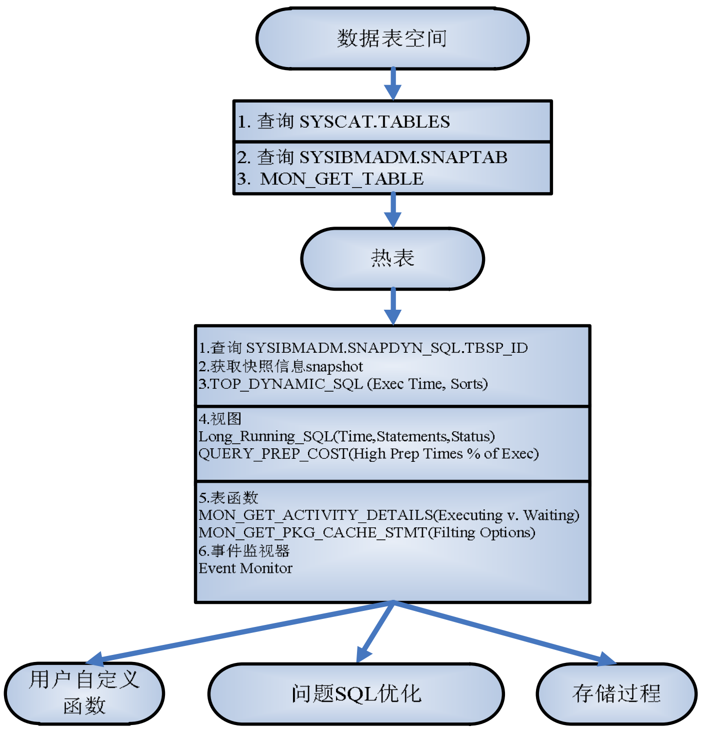


图 1-3　数据表空间性能优化树

2. 热索引导致的瓶颈

如图 1-4 所示的分析过程，检查繁忙设备所在的文件路径上都有哪些索引表空间，这些索引表空间中都有什么热索引，其中哪些索引是高度活跃的。在 OLTP 系统中，可以尝试下面的方法，例如增加缓冲池的大小，或为索引指定专门的缓冲池，从而降低 I/O 来消除瓶颈。但在 OLAP 环境中，索引通常非常大，分配足够的缓冲池来消除 I/O 不大可能，这种情况下，通过增加额外的容器以提高磁盘 I/O 并行度来消除瓶颈或许更加有效。

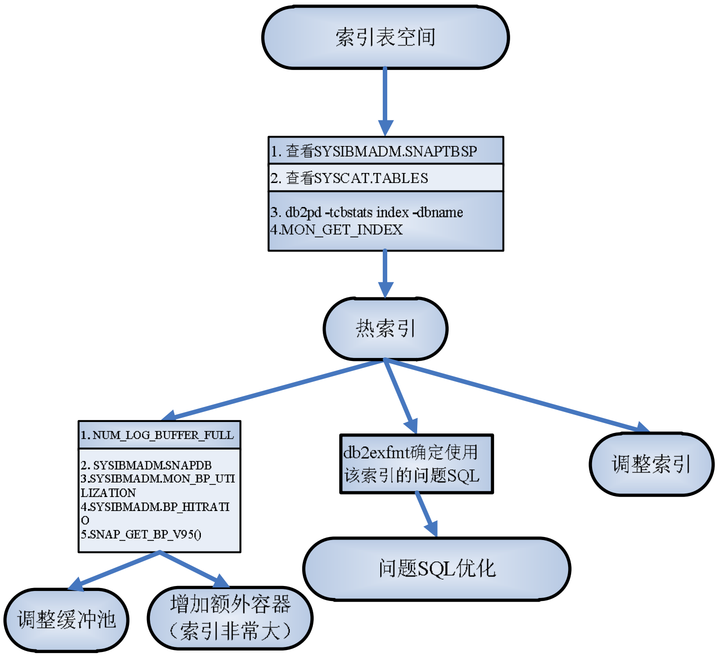


图 1-4　索引表空间优化树

3. 用户临时表的频繁读写导致的瓶颈

检查繁忙设备所在的文件路径上都有哪些用户临时表空间，这些用户临时表空间中都有什么频繁访问的临时表。由于排序溢出或者庞大的中间结果，导致对用户临时表的频繁读写进而造成磁盘瓶颈，分析诊断过程如图 1-5 所示。

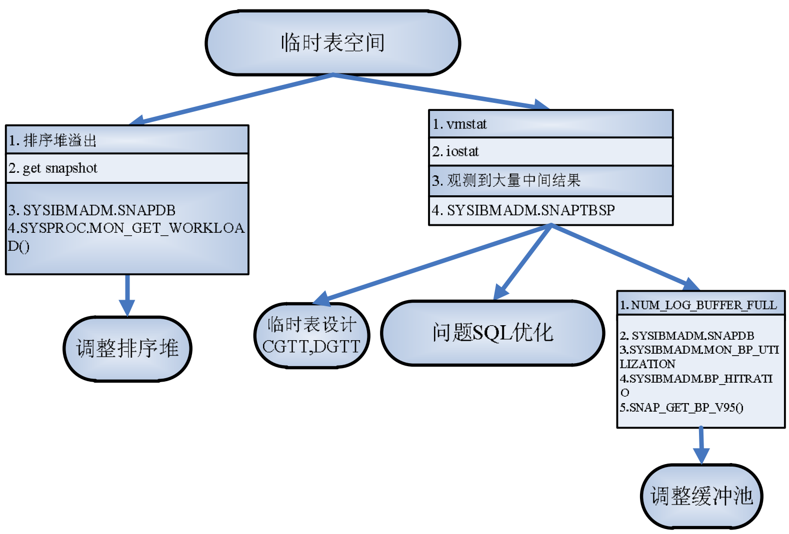


图 1-5　临时表空间容器优化树

4. 日志设备的频繁读写导致的瓶颈

检查繁忙设备所在的文件路径上是否用于存放事务日志。因为日志的读写速度能影响系统中所有的 INSERT/UPDATE/DELETE 语句，所以日志设备的读写速度对数据库性能有重大影响。

建议日志和数据要放在不同的物理盘上，避免出现磁盘争用。关于日志设备瓶颈的分析诊断过程如图 1-6 所示。

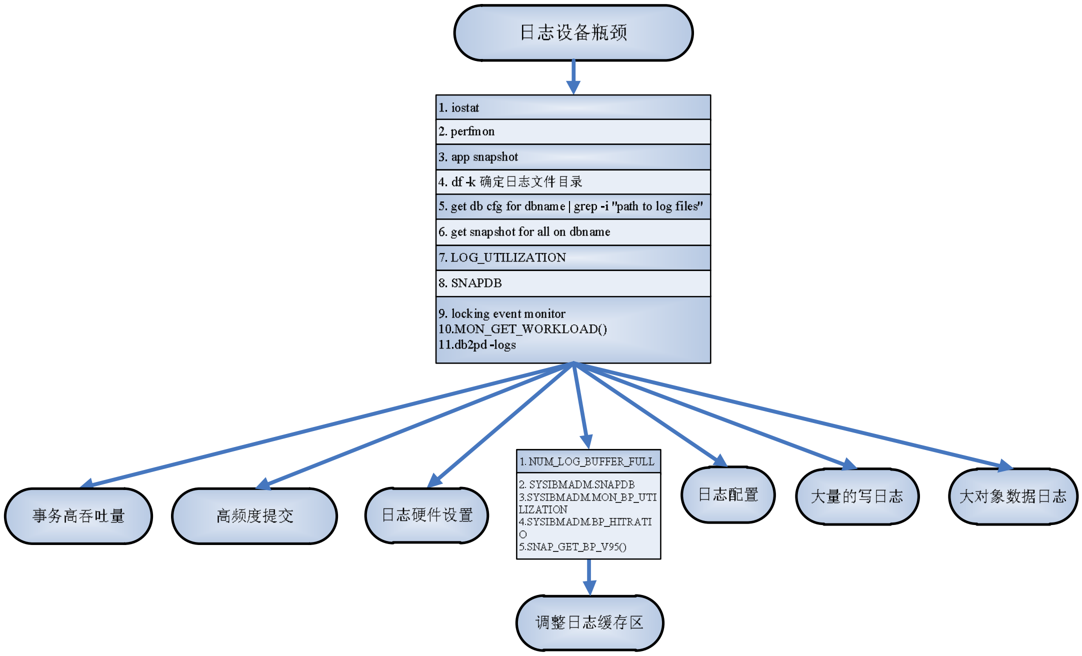


图 1-6　日志存储问题优化树

> **注意：db2diag.log 的频繁写入也会导致 IO 瓶颈**
>
> db2diag.log 的频繁写入也会造成 IO 瓶颈，特别是在多分区环境中，所有分区通过 NFS 文件系统共享诊断日志文件，这对性能的影响更大。解决这个问题最简单的办法是为每个分区指定一个单独的诊断日志路径和一个单独的诊断日志文件。

1.4.2　CPU 瓶颈

CPU 瓶颈表现在两个方面：用户态 CPU 瓶颈和系统态 CPU 瓶颈。运行操作系统内核以外的软件时导致的瓶颈为用户态 CPU 瓶颈，运行操作系统内核的时候导致的瓶颈为系统态 CPU 瓶颈。用户态 CPU 和系统态 CPU 时间比率在 3:1 到 4:1 之间是正常的。如果在有瓶颈的系统中，用户和系统时间比率高于这个区间，就应该分析用户态 CPU 时间增加的原因。

> **注意：CPU 并不是利用率越低越好**
>
> 很多读者觉得 CPU 利用率越低越好，这是不完全正确的，CPU 利用率应在一个合理范围内。在 OLTP 应用中，CPU 利用率越低，说明 CPU 有浪费，如果设计合理，应该能充分利用硬件资源，CPU 利用率越高就说明资源得到了最充分的利用，反而是件好事了。

1. 用户态 CPU 瓶颈

对用户态 CPU 瓶颈的诊断分析，如图 1-7 所示，主要从以下四个方面去分析。

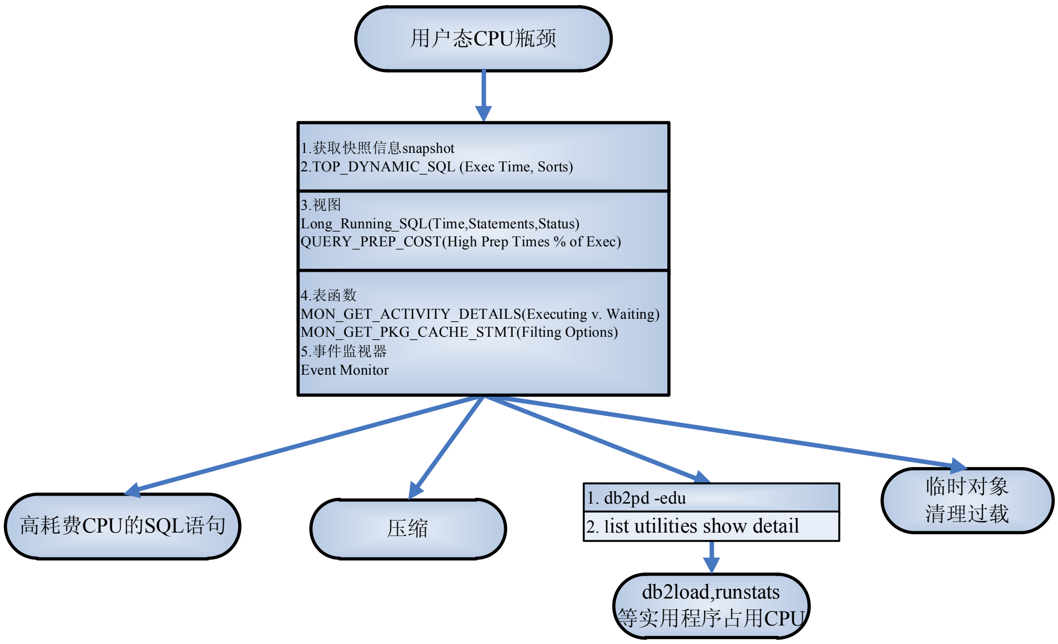


图 1-7　用户态 CPU 瓶颈优化树

（1）首先通过监控工具找到对应的动态 SQL 语句和静态 SQL 语句，随后从以下方面进行优化：

- 避免对大表进行频繁的表扫描，这样会消耗大量的 CPU 时间，建议创建合理的索引。
- 许多应用程序通过字符串拼接 SQL 语句查询条件，这会消耗 CPU 时间，建议使用参数标记。
- 如果应用程序执行一个 SQL 语句却只会用到一小部分产生的数据，建议使用 OPTIMIZE FOR n ROWS（OFnR）和 FETCH FIRST n ROWS ONLY（FFnRO）子句，这样可以减少 CPU 消耗。

（2）如果对表启用压缩，那么会消耗 CPU 时间。

（3）在一个实用工具正在运行时，CPU 使用率会有显著的增加。Load 和 runstats 就是很好的例子，它们经常导致 CPU 利用率变高。

（4）当删除临时表的时候，DB2 必须从缓冲池中移除不再需要的页面。如果此操作经常发生，并且临时表和普通用户数据共享缓冲池，会耗费额外的 CPU 时钟周期来解决冲突。

2. 系统态 CPU 瓶颈

在大多数 CPU 受限的环境中用户 CPU 往往是决定因素，不过系统 CPU 有时也能成为决定因素。如图 1-8 所示，系统 CPU 瓶颈可能由以下几方面造成：

（1）操作系统上下文切换率过高。当上下文切换得太频繁，会消耗大量 CPU 时间。导致上下文切换率高的原因可能是有大量的 DB2 连接。

（2）设备中断也会导致 CPU 时间高。一次中断的成本并不高，然而如果中断的频率太高，总的负载会非常高。

（3）DB2 数据库占可用内存的 50%左右通常比较合适，如果分配了过多的内存给 DB2 从

而导致操作系统可用内存不足，这样页清除操作将造成系统 CPU 开销。

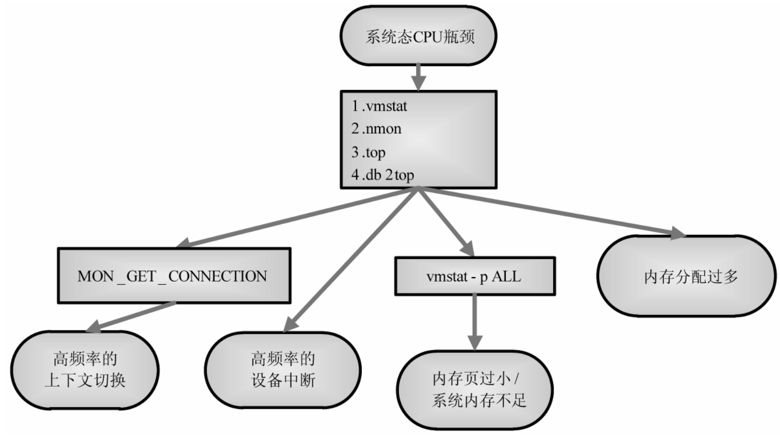


图 1-8　系统态 CPU 瓶颈优化树

1.4.3　内存瓶颈

如果没有充足的内存，数据在填满缓冲后就将转向 I/O，在这个过程中常常也会造成磁盘瓶颈。另外，内存也被用来存储元数据和运算结果，比如 SQL 查询计划和锁。如果缺少内存，系统不得不取消或销毁那些重要信息，并在稍后需要时重新计算，从而增加了 CPU 的开销。因此，有时候一个内存瓶颈的表现形式可能是磁盘或 CPU 问题。

为了分析和诊断内存瓶颈，可用从以下方面入手，如图 1-9 所示：

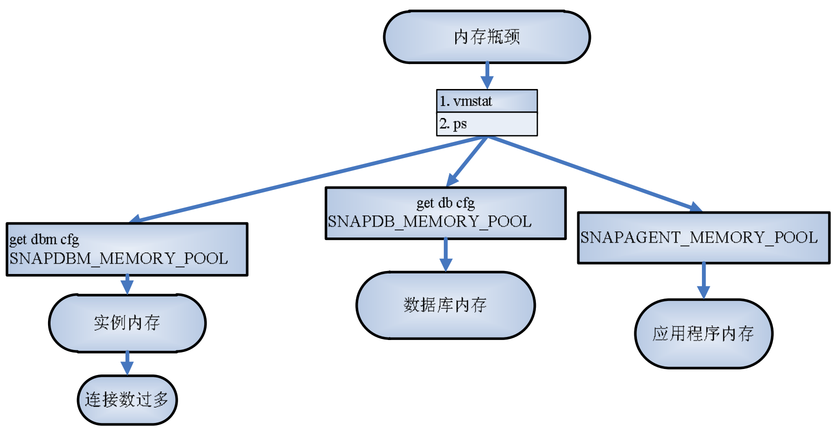


图 1-9　内存瓶颈优化树

（1）实例级和数据库级：主要是共享内存，包括缓冲池、排序堆、锁列表、包缓存等。共享内存受到 INSTANCE_MEMORY 和 DATABASE_MEMORY 配置参数的限制，它和数据库连接数没有关系。

（2）连接层面或应用程序级分配：包括应用程序堆和语句堆，内存消耗明显依赖于连接数。

无论是否启用 STMM，判断 DB2 内存实际用量的最好办法就是查看数据库管理器、数据库和应用程序快照的内存使用部分，它们提供了实例、数据库和应用各自的内存配置大小和当前实际内存大小。

1.4.4 网络瓶颈

如图 1-10 所示，通常网络方面的性能问题主要表现在以下三个方面：

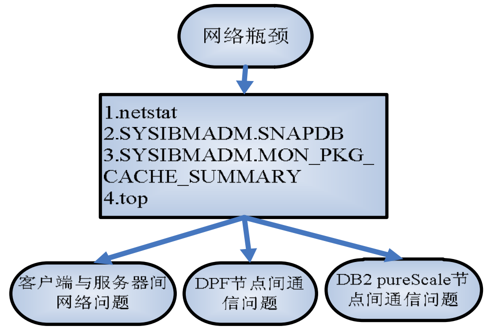


图 1-10 网络瓶颈种类

（1）客户端和服务器之间网络问题。例如业务数据量达到网络所能允许的峰值，或者服务器端返回客户端并不需要的大量数据等。

（2）在 DPF 多分区环境下，不同节点之间通信的问题。例如，不同节点之间通信数据量过大，导致网络延时，响应时间变慢等。

（3）在 DB2 pureScale 集群环境中，CF 节点和成员节点之间需要交互大量数据，所以推荐使用 Infiniband 网络或者万兆网，如果网络存在性能瓶颈，就会导致严重的性能问题。

1.4.5 懒惰系统

还有一种“懒惰系统”的性能问题，没有明显的 CPU、磁盘、内存或网络瓶颈，系统也不忙。如图 1-11 所示导致懒惰系统的几种可能性，现在分别叙述一下。

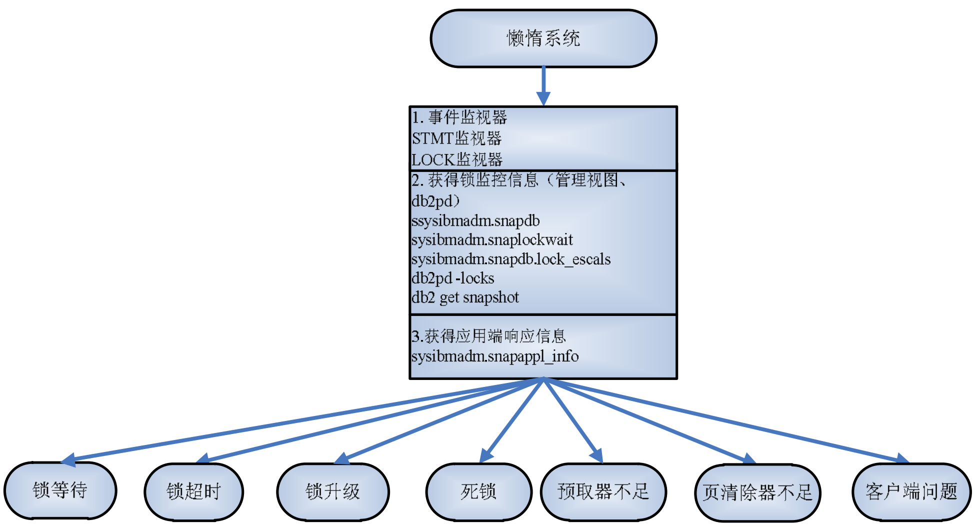


图 1-11 懒惰系统优化树

（1）在“懒惰系统”中一个最常见的原因是锁等待。锁等待是很容易在快照数据中被检查到的。查询管理视图 sysibmadm.snapdb 的字段 lock_wait_time 和 locks_waiting，可以看到锁等待时间和当前正在等待锁的代理进程数。

（2）为了防止请求锁的应用程序无限期等待，DB2 提供了 LOCKTIMEOUT 数据库参数，当锁等待的时间超过这个参数的设定值，就会返回出错消息，并回滚事务，这就是锁超时。当 LOCKTIMEOUT 的值设置为 - 1（这也是缺省值）时，请求锁的应用程序会无限期等待。

（3）锁升级也是“懒惰系统”中一个常见现象，锁升级导致的块或表级别的锁会产生严重的性能问题。

（4）另外还有死锁也会导致“懒惰系统”。死锁和应用程序有很大关系，从数据库层面监控和诊断，最终结合应用程序来解决。

（5）DB2 预取器、页清除器数目过小，也是导致懒惰系统的一个重要原因。

（6）DB2 服务器运行正常，但是由于客户端应用程序编写的质量问题，有时候也会导致懒惰系统。

从上面看出，大部分时候是由于锁问题导致的，关于这块具体内容请参考本书第 8 章相关章节。

1.4.6 关键 SQL 语句优化

通过对上述性能瓶颈的深入分析，很多读者会发现应用中包含的一些关键 SQL 语句往往是性能瓶颈的根源。下面我们来简单探讨一下如何对关键 SQL 语句进行优化。

注意：优化 SQL 语句，关键是分析其访问计划

很多读者可能会认为，优化 SQL 语句的性能，就是如何编写 SQL 语句，这种理解不全面。因为优化 SQL 语句，修改语句只是其中一种方式，优化的关键是分析和调整其访问计划。

如图 1-12 所示，关键 SQL 语句包括动态语句和静态语句，其优化方法包括手工方式和 Design Advisor 工具方式。手工方式需要靠 DBA 自己分析查询计划，自己使用索引或者表分区技术优化 SQL 语句；Design Advisor 工具方式的好处在于方便快捷，它可以自动给出专家建议，包括设计索引、设计表分区、设计 MQT 或者设计 MDC。

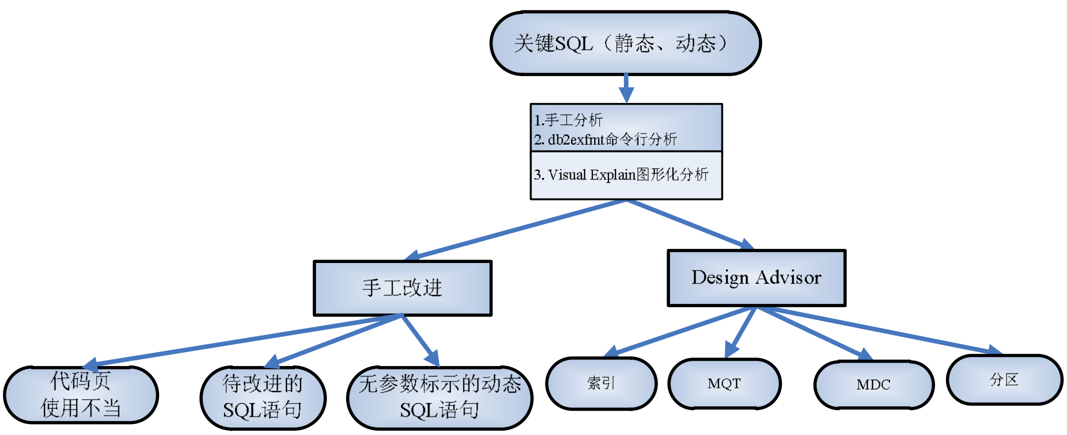


图 1-12 关键 SQL 语句优化树

关于 SQL 语句优化的技术，请参阅本书第 9 章的相关内容。

1.5 性能优化基本准则

让 DB2 数据库保持良好的性能，这并不是一件容易的事情。根据我们的优化经验，特给出

以下基本准则，供读者参考。

（1）20/80 原则也适用于性能优化

随着应用和数据库越来越大，全面调整应用变得代价高昂，所以，20/80 原则的运用显得尤为重要。这是因为，80%的数据库性能问题是由 20%的应用导致的，换句话说，只要优化这 20%的关键应用部分，就能解决 80%的问题。

（2）对 OLTP 应用和 OLAP 应用要区别对待

性能优化目标和应用的种类是紧密相关的，不同的应用种类要采用针对性的技术运用策略。针对 OLTP 应用，尽最大努力提升系统的响应速度是首要的优化目标，而对 OLAP 应用，则以整体数据吞吐量为目标。

（3）数据库的设计对数据库性能有至关重要的影响

在逻辑设计过程中，一定要确保用最合理的数据库技术来实现业务逻辑，例如表、索引、表分区、数据库分区、压缩等。在物理设计时，要合理地利用硬件资源，这就要求要有良好的存储设计、缓冲池设计、表空间设计等。

（4）应用开发的质量也会对数据库性能产生重要影响

一定要重视应用开发的质量，要确保应用持有锁的时间尽量短；要对 DB2 优化器有一定程度的了解，这样可以通过分析 SQL 语句的访问计划来判断 SQL 语句性能是否高效；要特别重视索引的运用，因为有时候一个合理的索引能对查询语句有极大的性能提升。

（5）合理配置系统参数和数据库参数

合理配置参数是性能提升的必要条件，所以要务必保证参数得到合理配置。

（6）运用维护窗口做好数据库维护工作

生产库在运维中，都会有维护窗口供使用。所以，在维护窗口期间，要及时执行 runstats，以使得优化器有最新的统计信息；也要及时执行 reorg，这样当有频繁的插入、删除和更新操作后，数据能得到重新整理从而提高性能。

1.6 小结

本章首先提出了性能问题，随后基于笔者在实际案例中的性能优化实践，提出了性能优化方法学，即自上而下方法学和自下而上方法学。

如果要建设一个新的数据库系统，那么必然会经历设计、开发和维护的各个阶段，按照自上而下方法学，在考虑业务功能实现的同时，性能优化应贯彻始终，越早越好，因为开始的越早，其付出的成本也就越小。

如果数据库系统已经处于维护阶段了，这时遇到了性能问题，那么使用自上而下方法学肯定是来不及了，只能采用自下而上方法学。按照自下而上方法学，本章讲解了 5 种类型的性能瓶颈，即磁盘瓶颈、CPU 瓶颈、内存瓶颈、网络瓶颈和懒惰系统，并给出了相应的监控和分析方法。最后，与大家分享了性能优化的基本准则。

第 2 章 实战案例研究与分享

某汽车制造商的 ERP 应用系统出现了性能问题，经过调查发现，要想彻底解决这个问题，就需要对支撑该 ERP 应用的 Oracle RAC 数据库进行架构改造，但是成本高昂，无法满足客户降低总体拥有成本的需要。这时，我们提出的“分库”架构方案得到了该客户的高度认可。本章基于此方案，详细讲述 DB2 经营分析库和 DB2 HADR 灾备库的实施过程，中间也穿插分享 DBA 需要重视的物理设计、逻辑设计等关键问题。最后，介绍通过自下而上方法学如何成功解决这个系统上线一段时间后遇到的性能问题。

让我们开始这个跌宕起伏的实战之旅吧。

2.1 性能问题的提出

我们优化过很多数据库系统，刚开始面临的状况几乎都是一样的：当接到客户报告性能问题时，往往是问题已经比较严重了，而且已经影响了业务的正常运转，这个时候客户处于非常焦急但又束手无策的状态，现在这个汽车制造商的性能问题也不例外。

2.1.1 系统现状：从 Oracle RAC 谈起

如图 2-1 所示，这个数据库系统基于典型的 Oracle RAC 集群架构，选择 Oracle RAC 是为了高可用性和负载均衡的需要，Oracle 数据库的版本为 11g，包括两个节点，一个是节点 0，另一个是节点 1，这两个节点各自运行在独立的 Power 740 服务器上，其中硬件配置为 16 个 CPU 内核，64G 内存，操作系统为 AIX 6.1；共享存储使用 IBM V7000，SAN 交换机将该存储和节点 0、节点 1 互联；网络方面，私网交换机（千兆）实现双节点的内部互联，公网交换机（千兆）实现双节点和 ERP 应用的外部互联。

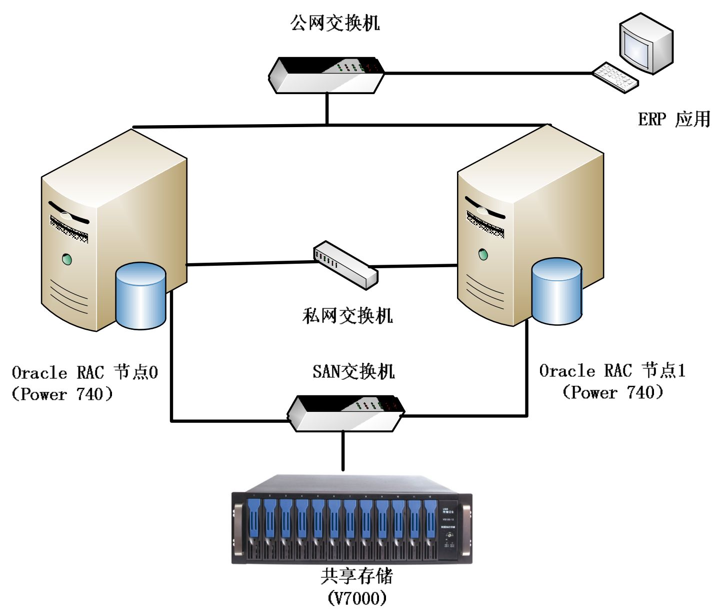


图 2-1 某汽车制造商 ERP 系统（基于 Oracle RAC）

从工作负载来看，对数据库的操作主要是读与写并发，读写比平均为 7:3，平常并发连接为 500 个，高峰时达到 2000 个左右。这个系统的负载包括下面三种：

（1）业务流程负载：供业务人员使用，涵盖了供应链、生产、财务、营销等各个模块，这些模块总体上的负载以事务型为主，这对数据库系统的事务吞吐量提出了非常高的要求。

（2）查询分析类负载：供管理人员使用，例如企业运营情况查询、人力资源管理等，这类用户对响应时间有严格的要求。这类用户可对数据进行分析以获得透视能力，并在此基础上采取行动。用户可以访问、计算及共享信息，以便跨越产品、市场、职权、流程、时期和情景等进行绩效检查、质量评判以及因果分析。

（3）报表类负载：包括月报表、季报表、半年报表，年报表等，通常在每个时间单位的最后几天进行，例如月报表在每个月的 25 号开始，在 30 号左右完成，这对生成时间有极高的要求。

这个 ERP 系统是分期上线的，刚开始只有供应链、生产和财务模块，后来逐渐增加了营销、人力资源等模块；用户量也从刚开始的平时 100 个并发用户逐渐增加到 500 并发用户，高峰时可以达到 2000 左右；数据量的增加更为显著，从上线初期的 1TB 增加到现在的 6TB，由于写负载比较频繁，每天变化的数据量可以达到 200G 左右。这样一来，这个汽车制造商 ERP 应用的 Oracle 数据库性能就开始遇到了瓶颈，系统资源不堪重负，具体表现在以下几个方面：

（1）业务流程管理中各模块包括供应链、生产、财务、营销和人力资源等需要处理的事务量越来越多，特别是系统繁忙时，已经接近集群对事务吞吐量的设计上限。

（2）分析查询性能出现了问题，按照设计，只能提供按天或者按月的查询，但是根据动态条件查询上存在性能问题，例如要求按照某个时间段查询，需要 5 分钟以上的时间才能访问结果，而前端的应用逻辑是 2 分钟数据库没有返回结果就认为是超时了。

（3）报表应用也存在性能问题。目前生成报表的时间在 30 个小时左右，随着数据量的不断增长，这个时间还在逐渐增加。从性能角度来看，应该寻找一种不完全依赖于数据量的方案。

需要强调的是，上述不同种类的负载出现性能问题后，会出现严重的叠加效应，特别是交易型的流程管理和分析型的日常查询、报表类应用在执行时间上难以避免出现了重叠，这使得数据库不堪重负。

因此，一个非常棘手的性能问题就摆在了这个汽车制造商的面前，到底应该怎么解决呢？

2.1.2 性能优化解决方案："分库"架构

上一节，我们了解了该汽车制造商 ERP 应用的 Oracle RAC 集群遇到了性能问题，这一节我们谈谈针对这个问题的解决办法。

在第一章讲过成本效益分析，这个策略是用来找到成本和性能的最佳平衡点的。性能优化并不意味着片面追求性能的最大化，因为优化是无止境的，必须根据业务需求找到真正的痛点所在，其实，这个问题最大的痛点是 ERP 应用的不同负载的叠加导致的，谈起这个问题，很多读者可能会自然想到下面的两种办法：

第一种办法，立足于目前的软硬件条件，对 Oracle RAC 数据库集群进行优化。通常的步骤是，首先对操作系统和数据库进行监控，随后调整参数，调整物理设计，调整逻辑设计诸如索引、表分区、临时表等技术，最后再进行性能验证。这种办法的优点是不需要对数据库进行大的架构调整就能实现，但是，缺点也很明显，这种办法依赖于优化人员的经验和技术水平，通常只对局部应用优化有帮助，对整体性能往往无能为力，所以是一种权宜之计。另外，有时候也会出现投入产出不成正比的问题，投入足够多的人力，但是优化效果却不尽如人意。

第二种办法，硬件扩容的方式。具体来说，就是将集群从 2 个节点扩充为 4 个节点来提升系统的处理能力。这种办法的优势是可以非常明显地提升系统的事务吞吐能力，也能提升 ERP 应用中某些模块的性能。但是，运行 Oracle RAC 的 Power 740 服务器将从 2 台变为 4 台，而 Power 740 价格相对较高，这超过了该汽车制造商的硬件预算。最重要的是 Oracle RAC 的强项在于高可用性和负载均衡，但是扩展能力不是线性的，另外设计上也需要进行必要的改造，例

如对一些热点表进行 hash 分区或者范围分区，否则数据在多个节点之间多次传递影响性能。

那么，还有没有更好的办法呢？本章将向读者介绍一种分布式架构方案，即对硬件“分库”的解决方案，具体来说，就是根据应用负载的不同而将单一的数据库分解成同构或者异构的数据库。

比较：集中式架构和分布式架构

数据库架构分为集中式和分布式。单机版的 DB2 以及 DB2 PureScale 集群都属于集中式的。分布式架构的产品和技术很多，例如异构数据库实时复制产品 IBM CDC、灾备技术 DB2 HADR 等。

在本客户案例中，可以按照不同的负载方式对这个 ERP 应用进行“分库”处理：上面谈到供应链、生产、财务、营销等业务模块，他们的负载特点是事务型的，以写为主，所以这些应用运行在一个独立的数据库中；另外一种是分析查询和报表，包括上面谈到的分析企业运营情况、人力资源管理等，这些负载的特点是分析型的，以读为主，这些应用运行在另外的数据库中。这样“分库”之后带来的好处是可以实现读写分离，逻辑上也更加清晰，能从架构上非常明显地提升系统性能。

注意：架构与性能

数据库架构规划和设计应该满足短期、中期和长期的性能需求，但是由于业务的快速发展，迟早有一天会产生性能问题，这样将会对数据库架构提出严重挑战，从而促使架构不得不与时俱地进持续改变，以再次满足业务的性能需求。

图 2-2 是对事务型负载和分析型负载“分库”后的架构图：Oracle RAC 集群还是基于原有的硬件环境，不过进行了“瘦身”处理，只用来运行事务型负载，包括供应链、生产、财务等；考虑到 DB2 数据库在运行分析型负载的性能优势，所以选择了异构的 DB2 数据库来运行分析型应用，包括分析企业运营情况、人力资源管理等，运行 DB2 的硬件选择性价比更高的 x3850 服务器，操作系统为 Redhat Linux，配置了 64 个 CPU 内核、256GB 内存。

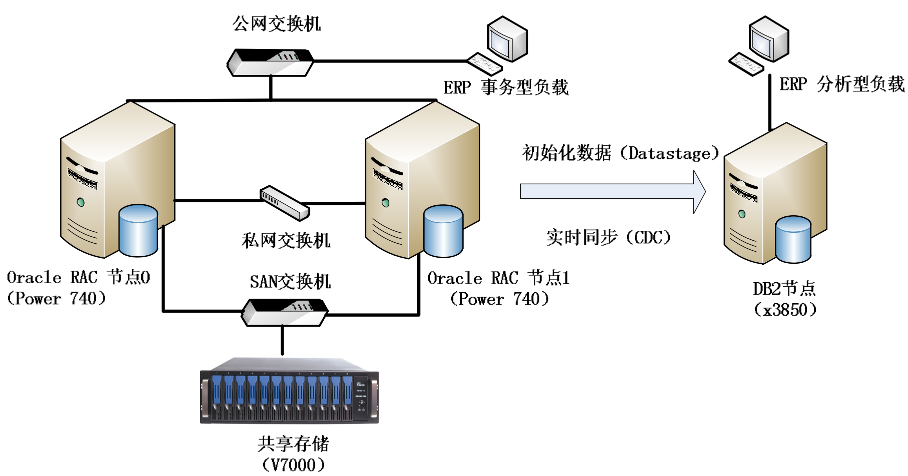


图 2-2　“分库”后的架构图

按照上述方式“分库”后，产生了新的问题。因为分析型负载应用需要基于和 ERP 事务型负载应用逻辑上一致的数据，换句话说，要解决好下面的两个问题：

（1）目前 Oracle 数据库中的数据量已经达到了 8TB，那么如何将这样一个 TB 级的数据初始化到 DB2 数据库中？

（2）数据初始化后，如何保证 Oracle 源数据库和 DB2 目标数据库之间保持数据实时同步？

经过探讨，解决上述问题的办法是借助其他数据整合工具来完成，TB 级的数据初始化需要借助 ETL 工具，例如 IBM Datastage 或者 Oracle ODI；数据库之间实时复制需要借助 IBM CDC 或者 Oracle GoldenGate。于是，分别对这两套工具进行了技术验证（Proof Of Concept）。

比较：IBM Datastage 和 Oracle ODI

验证结果表明，从 Oracle 到 DB2 迁移同样规模的数据，IBM Datastage 可以实现平均 10 万条/秒以上记录的迁移，而 Oracle ODI 只能达到 5 万条/秒。

比较：IBM CDC 和 Oracle GoldenGate

IBM CDC 在安装的简便性及图形化界面操作方面，要比 Oracle GoldenGate 好很多，而且当目标端是 DB2，并且需要每天同步 200G 以上变化数据时，IBM CDC 表现出非常高的可靠性和性能。

最终，该汽车制造商选择了 IBM Datastage 工具实现从 Oracle 到 DB2 之间的数据初始化，选择了 IBM CDC 来实现 Oracle 和 DB2 之间实时同步。

2.1.3　高可用解决方案：DB2 HADR

上一节我们提出了一种针对不同负载“分库”的方案来解决 Oracle RAC 集群的性能问题，其中引入了 DB2 节点来分担（Off Load）分析查询类负载。但是，这个汽车制造商并没有满足于此，而是提出了一个这样的问题：当初选择 Oracle RAC 就是为了实现高可用性，以满足数据库系统 7×24 小时不间断运行的需要，而企业运营情况查询、人力资源管理等查询分析业务，也需要高可用和灾备方案，那么在 DB2 中如何实现呢？答案是“DB2 HADR”。

可能很多读者第一次听到 DB2 HADR 这个名字，那么什么是 DB2 HADR 呢？它是 DB2 High Availability Disaster Recovery 的英文缩写，如图 2-3 所示，它是一种实现高可用性和灾难恢复的解决方案。DB2 HADR 由主数据库和从数据库组成。主数据库接收连接并且执行事务，将生成的日志实时地发送至从数据库；从数据库将主数据库发送过来的日志进行重做，从而完成与主数据的同步。当主数据库出现问题不能正常工作时，从数据库可以快速接管主数据库的功能，DBA 可用选择手工接管或者自动接管。

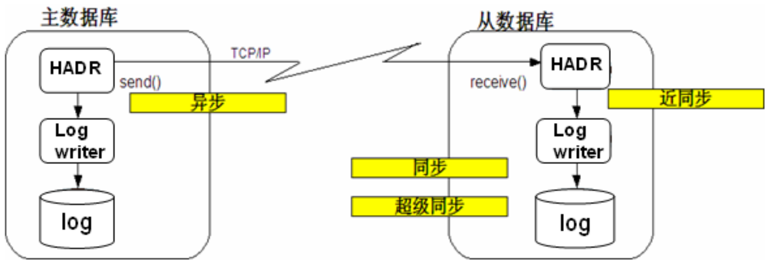


图 2-3 DB2 HADR 原理

比较：DB2 HADR 的四种同步模式

DB2 HADR 支持四种同步模式：同步、近同步、异步和超级异步。四种同步模式对网络的要求和对主数据库的影响依次减小，但是可靠性也依次降低。例如，同步模式对主数据库上的事务影响最大，但是可以保证从数据库上的事务和主数据库是实时一致的；而超级异步模式对主数据库上的事务没有任何影响，但是一旦主数据库出现事故，就可能会发生事务丢失。

引入 DB2 HADR 后，最终经营分析库如图 2-4 所示，包括 DB2 主节点和从节点，服务器都为性价比极高的 x3850 服务器，64 个内核，256GB 内存。为了真正确保高可用性，DB2 主节点和从节点是跨机房部署的，其中主节点部署在北京大兴机房，从节点部署在北京亦庄机房，两个节点之间通过 VPN 网络连接，选择近同步工作模式。

为了使用好 DB2 HADR，还需要在客户端配置自动客户端重新路由（Automatic Client Reroute，缩写为 ACR）功能，这样当主节点和从节点之间发生故障切换时，可以将 ERP 查询分析应用自动路由到从节点上，这可以最大限度地减少中断时间。

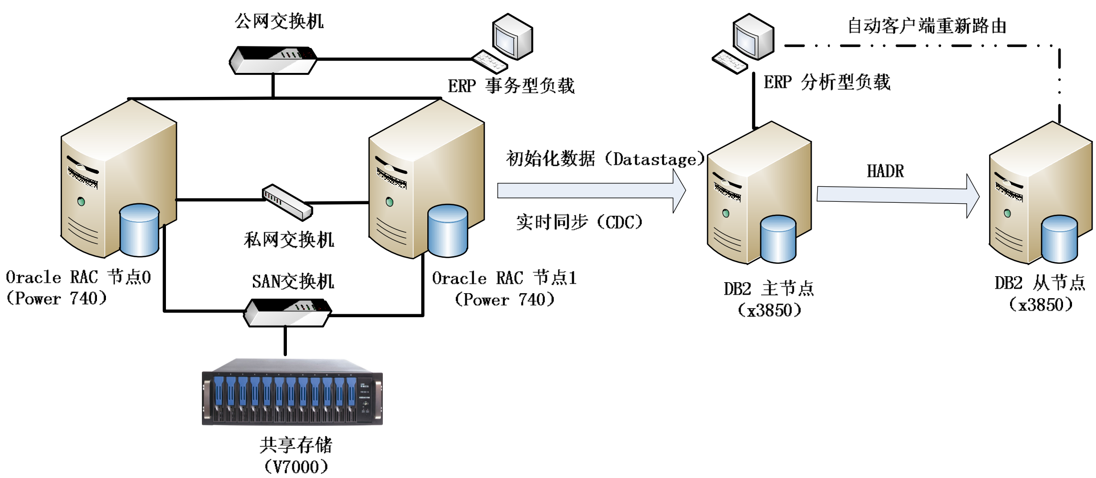


图 2-4 引入 DB2 HADR 后的架构图

从 DB2 V9.7 开始，DB2 HADR 开始支持从节点可读（Read On Standby）的功能，用户可以将一些读操作集中在从节点上运行，以减轻主机的负载。另外，从 DB2 V10 开始支持 3 个从节点。

在这个 ERP 案例中，DB2 HADR 的优势可以得以利用：配置、使用和管理非常简单；近同步模式的管理非常有效，丢失数据的可能性非常小；主节点和从节点之间只传输日志，节省了网络上的开销；主从数据库的数据几乎实时同步，接管的速度也非常快，极大地减轻了数据库管理员的工作。

2.2 实施：经营分析库

这一节将从性能的角度谈谈基于 DB2 的经营分析库的实施过程，将分享实施中务必重视的物理设计、逻辑设计、参数配置等，这将为后续的性能优化打下坚实的基础。这次实施，选择了 DB2 V9.7 FP5 作为安装介质，这是一个可靠和稳定的版本。

2.2.1 DB2 数据库安装

DB2 安装程序提供三种方式用以安装 DB2 数据库软件。一种是图形化界面方式，一种是命令行方式，另一种是静默安装方式。每种安装方式，都有适合的场景和特点。例如，图形化界面的安装方式，适合习惯使用界面交互方式的用户使用；命令行安装方式，适合具备一定 DB2 使用经验的用户，在无法使用图形界面的环境中轻松安装 DB2 数据库软件；静默安装方式，整个安装过程无须人工干预，通过响应文件的方式，可以方便用户在多台服务器上，同时安装相同配置的 DB2 数据库软件。

本次实施中，该汽车制造商客户出于安全方面的考虑，另外也为了避免消耗不必要的资源，禁止使用图形界面，所以我们只能使用命令行方式安装，这种安装方式，其实也很方便、快捷。接下来，介绍一下整个安装过程。

1. 创建用户和组

按照 DB2 的安装要求，如表 2-1 所示，一共规划了三个组，erpiadm1（实例用户组）、erpfadm1（防护用户组）和 dasadm1（管理服务器用户组），这三个组中分别包括三个用户，分别为 erpinst1（实例用户）、erpfenc1（防护用户）和 dasusr1（管理服务器用户）。需要注意的是，在创建上述组和用户时，需要显示指定组 ID 和用户 ID，这是因为搭建 DB2 HADR 环境时，要求主节点和从节点相应的组 ID 和用户 ID 完全一致。关于 DB2 HADR 双机的实施，可参见 2.3 节有关内容。

注意：DB2 管理服务器已不推荐使用

DB2 管理服务器（DB2 Administrative Server，缩写为 DAS）从 DB2 V9.7 开始已不推荐使用，所以相应的 dasadm1 和 dasusr1 不是必须要创建的。

表 2-1 用户和组规划表

用户名/组名	ID	分类	根目录	用途
erpiadm1	600	组	-	实例用户组
erpfadm1	601	组	-	防护用户组

(续表)

用户名/组名	ID	分类	根目录	用途
dasadm1	602	组	-	管理服务器用户组
erpinst1	600	用户	/db2home/erpinst1	实例用户
erpfenc1	601	用户	/db2home/erpfenc1	防护用户
dasusr1	602	用户	/db2home/dasusr1	管理服务器用户

下面是在 Redhat Linux 上创建组、用户和设置用户密码的命令：

（1）创建组

```
groupadd -g 600 erpiadm1
groupadd -g 601 erpfadm1
groupadd -g 602 dasadm1
```

（2）创建用户，并指派用户组

```
useradd -u 600 -g erpiadm1    -m -d /db2home/erpinst1 erpinst1
useradd -u 601 -g erpfadm1 -m -d /db2home/erpfenc1 erpfenc1
useradd -u 602 -g dasadm1 -m -d /db2home/dasusr1 dasusr1
```

以上述第一条 useradd 命令为例，**-g** 指定了用户的初始组（initial_group）；**-m** 指定了如果用户目录不存在，将自动创建；**-d** 指定了用户根目录；erpinst1 是创建的实例用户名，erpfenc1 是创建的防护用户名，dasusr1 是创建的管理服务器用户名。

（3）设置用户密码

```
passwd erpinst1
passwd erpfenc1
passwd dasusr1
```

2. 正式安装

把 DB2 安装介质拷贝到 X3850 目标机器上，进入 DB2 安装介质存放路径，运行脚本 db2_install，看到选择安装目录的命令提示，采用默认路径/opt/ibm/db2/V9.7。输入“ESE”，按回车键，安装 DB2 企业服务器版。安装程序显示开始初始化 DB2 安装过程，并提示共有 47 个任务需要执行以及所需的时间，等待一会儿，安装任务结束。

呵呵，稍安毋躁！到现在为止，仅完成了将 DB2 的安装文件拷贝到服务器上，还需要进行操作系统参数配置、实例创建和配置、数据库创建和配置等操作。

2.2.2 操作系统参数配置

1. 修改 sysctl.conf 内核参数文件

编辑/etc/sysctl.conf 文件来修改必须调整的内核参数。如果此文件不存在，那么应创建它。

```
# cat /etc/sysctl.conf
--下面是内存为 16GB 的配置例子
--如果内存多于 16GB, SEMMSL、SEMMNS、SEMOPM、msgmax 和 msgmnb 保持不变
--但是，shmmni、shmmax、shmall、SEMMNI 和 msgmni 的设置成比例增大，见下面加黑内容：
```

```
kernel.shmmni=4096
kernel.shmmax=17179869184
kernel.shmall=8388608
--kernel.sem=<SEMMSL> <SEMMNS> <SEMOPM> <SEMMNI>
kernel.sem=250 256000 32 4096
kernel.msgmni=16384
kernel.msgmax=65536
kernel.msgmnb=65536
```

x3850 服务器拥有 256GB 内存，是 16GB 的 16 倍，shmmni、shmmax、shmall、SEMMNI 和 msgmni 的设置应成比例增大，最终内核参数的配置如下：

```
# cat /etc/sysctl.conf
--内存为 256GB
kernel.shmmni=65536
kernel.shmmax=274877906944
kernel.shmall=134217728
--kernel.sem=<SEMMSL> <SEMMNS> <SEMOPM> <SEMMNI>
kernel.sem=250 256000 32 65536
kernel.msgmni=262144
kernel.msgmax=65536
kernel.msgmnb=65536
```

运行 sysctl 时附带-p 参数，以便从缺省文件/etc/sysctl.conf 中装入 sysctl 设置：

```
# sysctl -p
```

运行 ipcs -l 命令以显示当前的内核参数设置。

```
# ipcs -l
     ------ Shared Memory Limits --------
     max number of segments = 65536              // SHMMNI
     max seg size (kbytes) = 274877906944        // SHMMAX
     max total shared memory (kbytes) = 134217728 // SHMALL
     min seg size (bytes) = 1
     ------ Semaphore Limits --------
     max number of arrays = 65536               // SEMMNI
     max semaphores per array = 250            // SEMMSL
     max semaphores system wide = 256000        // SEMMNS
     max ops per semop call = 32               // SEMOPM
     semaphore max value = 65536
     ------ Messages: Limits --------
     max queues system wide = 262144             // MSGMNI
     max size of message (bytes) = 65536        // MSGMAX
     default max size of queue (bytes) = 65536    // MSGMNB
```

2. 修改 limits.conf 文件

编辑 limits.conf 文件，针对用户 erpinst1 修改进程的最大数目（nproc）、打开文件的最大数目（nofile）、最大栈大小（stack）、最大数据大小（data）、最大文件大小（fsize）的软限制和硬限制，最终为这台 x3850 服务器设置了如下参数：

```
# cat /etc/security/limits.conf:
...
```

```
erpinst1 soft nproc 2047
erpinst1 hard nproc 16384
erpinst1 soft nofile 4096
erpinst1 hard nofile 65536
erpinst1 soft stack 10240
-- -1 表示最大数据大小没有限制
erpinst1 hard data -1
-- -1 表示最大文件大小没有限制
erpinst1 hard fsize -1
```

3. 修改/etc/hosts 文件

实施中会用到两台 X3850 服务器，其中主节点 IP 地址为“10.2.3.1”，域名为“PrimaryNode-1”，从节点 IP 地址为“10.2.3.2”，域名为“StandbyNode-1”，如下所示：

```
# cat /etc/hosts
127.0.0.1            localhost.localdomain localhost
10.2.3.1          PrimaryNode-1
10.2.3.2          StandbyNode-1
```

2.2.3　实例创建与配置

完成安装和操作系统参数设置后，接下来规划数据库实例。如表 2-2 所示，数据库实例名和实例用户名相同，名为 erpinst1，实例的根目录为/db2home/erpinst1/sqllib；端口号设置为 50001，下面将讲述如何在/etc/services 文件中配置端口号；SVCENAME 设置为 db2c_erpinst1，下面将讲述如何在 DBM CFG 中配置。

表 2-2　数据库实例规划

数据库实例名	erpinst1
端口	50001
根目录	/db2home/erpinst1/sqllib
SVCENAME	db2c_erpinst1

1. 创建 DB2 管理服务器（可选）

如果在 2.2.1 节没有创建组 dasadm1 和用户 dasusr1 的话，就不需要再创建 DB2 管理服务器了，所以这个步骤可以省略。

如果需要创建的话，那么使用 root 用户，执行命令 dascrt，如果 DB2 安装在默认路径中，该命令的全路径为/opt/ibm/db2/V9.7/instance/dascrt，如下所示：

```
/opt/ibm/db2/V9.7/instance/dascrt -u dasusr1
SQL4406W The DB2 Administration Server was started successfully.
DBI1070I  Program dascrt completed successfully.
```

2. 创建 DB2 实例

使用 root 用户，执行命令 db2icrt，如果 DB2 安装在默认路径中，该命令的全路径为/opt/ibm/db2/V9.7/instance/db2icrt，如下所示：

```
/opt/ibm/db2/V9.7/instance/db2icrt -a server -u erpfenc1 erpinst1
DBI1070I  Program db2icrt completed successfully.
```

其中，-a 指定了实例采用服务器端(server)认证模式；-u 指定了受防护用户 erpfenc1；erpinst1 是创建的实例名。

只有在 DB2 服务器端配置了 TCP/IP 通信协议，DB2 服务器才能接受来自 DB2 客户端的请求，因此需要做下面的工作：

（1）使用 root 用户，在操作系统的 TCP/IP 服务文件（/etc/services）中指定服务器的监听端口 50001，内容如下：

```
db2c_erpinst1  50001/tcp # DB2 connection service port
```

（2）使用 DB2 实例用户 erpinst1，为数据库管理器配置参数 SVCENAME 赋值，命令如下：

```
db2 update dbm cfg using SVCENAME db2c_erpinst1
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command completed successfully.
```

（3）使用 DB2 实例用户 db2inst1，设置通信代理。

```
db2set DB2COMM=TCPIP
```

OK！大功告成！

2.2.4　存储规划与文件系统创建

在本次实施过程中，文件系统规划如表 2-3 所示，其中 db2home 使用 X3850 的内置盘存放实例数据。文件系统 db2fs1、db2fs2 和 db2fs3 使用外挂磁盘用于存放数据，其大小完全一致，底层物理规划也一样，即使用 2 个 RAID 5 盘组，每个 RAID 组为（4D+1P），每块盘大小为 300G。db2log 也使用外挂磁盘存放数据库日志，底层规划为 1 个（3D+1P）的 RAID 5 盘组。这样就可以达到数据和日志物理隔离的目的，从而提高数据库的 I/O 性能。

表 2-3　文件系统规划

文件系统	挂载点	容量	用途	磁盘
db2home	/db2home	300 GB	实例数据	内置盘
db2fs1	/db2fs1	2.4 TB	数据	单个物理盘为 300G， 2 个 RAID 5 盘组， 每个盘组为（4D+1P）
db2fs2	/db2fs2	2.4 TB	数据	单个物理盘为 300G， 2 个 RAID 5 盘组， 每个盘组为（4D+1P）
db2fs3	/db2fs3	2.4 TB	数据	单个物理盘为 300G， 2 个 RAID 5 盘组， 每个盘组为（4D+1P）
db2log	/db2log	900 GB	日志	单个物理盘为 300G， 1 个 RAID 5 盘组， 每个盘组为（3D+1P）

这个案例使用 LVM（Logical Volume Manager）来管理磁盘阵列。那么什么是 LVM 呢？LVM 是对磁盘分区进行管理的一种机制，它是建立在硬盘之上的一个逻辑层，用于提高磁盘管理的灵活性。

硬盘分区是 LVM 的基本存储逻辑块，称为物理卷（Physical Volume，简称 PV）。若干个 PV 组成一个整块的卷组（Volume Group，简称 VG），形成一个存储池。管理员可以在 VG 上随意创建逻辑卷（Logical Volumes，简称 LV），并进一步在逻辑卷上创建文件系统。创建文件系统 db2fs1 的过程如下，其中/dev/sdc 和/dev/sdd 对应的是存储映射过来的两个 RAID 5 盘组，使用“pvcreate”命令在它们上面分别创建物理卷；然后使用“vgcreate”命令创建 fs1vg 卷组，包括物理卷/dev/sdc，再使用“vgextend”命令把物理卷/dev/sdd 也加入到 fs1vg 卷组中；最后使用“lvcreate”命令在卷组 fs1vg 上创建大 小为 2400GB，也就是 2.4TB 的逻辑卷 fs1lv。

完成逻辑卷 fs1lv 的创建后，使用 mkfs.ext4 在其上创建 ext4 格式的文件系统，随后按照表 2-3 的规划，再用 mount 命令挂载，挂载点为/db2fs1。

```
pvcreate /dev/sdd
pvcreate /dev/sdd
vgcreate fs1vg /dev/sdc
vgextend fs1vg /dev/sdd
lvcreate -L 2400G -n fs1lv fs1vg
mkfs.ext4 /dev/fs1vg/fs1lv
mkdir  /db2fs1
mount /dev/fs1vg/fs1lv  /db2fs1
```

用类似的办法，在 RAID 5 盘组/dev/sde 和/dev/sdf 上创建文件系统 db2fs2，大小也为 2.4TB 左右，挂载点为/db2fs2，命令如下：

```
pvcreate /dev/sde
pvcreate /dev/sdf
vgcreate fs2vg /dev/sde
vgextend fs2vg /dev/sdf
lvcreate -L 2400G -n fs2lv fs2vg
mkfs.ext4 /dev/fs2vg/fs2lv
mkdir  /db2fs2
mount /dev/fs2vg/fs2lv  /db2fs2
```

用类似的办法，在 RAID 5 盘组/dev/sdg 和/dev/sdh 上创建文件系统 db2fs3，大小也为 2.4TB 左右，挂载点为/db2fs3，命令如下：

```
pvcreate /dev/sdg
pvcreate /dev/sdh
vgcreate fs3vg /dev/sdg
vgextend fs3vg /dev/sdh
lvcreate -L 2400G -n fs3lv fs3vg
mkfs.ext4 /dev/fs3vg/fs3lv
mkdir  /db2fs3
mount /dev/fs3vg/fs3lv  /db2fs3
```

用类似的办法，在 RAID 5 盘组/dev/sdi 上创建文件系统 db2log，大小为 900GB，挂载点为/db2log，命令如下：

```
pvcreate /dev/sdi
vgcreate logvg /dev/sdi
lvcreate -L 900G -n loglv logvg
mkfs.ext4 /dev/logvg/loglv
mkdir  /db2log
mount /dev/logvg/loglv  /db2log
```

为了方便每次系统重启后，文件系统 db2fs1、db2fs2、db2fs3 和 db2log 能够自动挂载，可以在/etc/fstab 文件的末尾加入下面的黑体语句，这样对 DBA 来说，就方便多了。

```
# cat /etc/fstab
devpts  /dev/pts         devpts  mode=0620,gid=5 0 0
proc    /proc            proc    defaults       0 0
sysfs   /sys             sysfs   noauto         0 0
debugfs /sys/kernel/debug debugfs noauto         0 0
usbfs   /proc/bus/usb    usbfs   noauto         0 0
/dev/fs1vg/fs1lv  /db2fs1 ext4    acl,user_xattr       1 1
/dev/fs2vg/fs2lv  /db2fs2 ext4    acl,user_xattr       1 1
/dev/fs3vg/fs3lv  /db2fs3 ext4    acl,user_xattr       1 1
/dev/logvg/loglv  /db2log ext4    acl,user_xattr       1 1
```

2.2.5 数据库创建

1. 数据库规划

这个汽车制造商的 DB2 经营分析库的名字为 erpdb，字符集选择 UTF-8，默认页大小为 32K，这样就和 Oracle 数据库的字符集和默认页大小完全一致了。

如表 2-4 所示，缓冲池的规划是按照应用来确定的。这个数据库主要运行人力资源管理、分析查询和报表三个应用，所以为这三个应用自定义三个独立的缓冲池；另外，为了方便管理和逻辑清晰，数据和索引使用不同的缓冲池。

小技巧：如何选择页大小

对于执行随机更新操作的 OLTP 应用，采用较小的页面例如 8K 或者 16K 更好，这样可以提高并发能力，因为较小的页面消耗的缓冲池空间小；对于要一次访问大量连续记录的 OLAP 或者 DSS 应用，采用较大的页面例如 16K 或者 32K 性能更好，这样读取特定数量的记录时发出的 I/O 请求会更少。

最终的规划是这样的：数据缓冲池 HR_DBP 和索引缓冲池 HR_IBP 为人力资源模块使用，由于人力资源模块对数据库的访问以随机访问为主，它们的页大小都规划是 8KB；数据缓冲池 QRY_DBP 和索引缓冲池 QRY_IBP 为分析查询模块使用，由于分析查询模块对数据库的访问既有随机访问，也有大片连续访问，它们的页大小规划为 16K；数据缓冲池 RPT_DBP 和索引缓冲池 RPT_IBP 为报表模块使用，由于报表模块对数据库的访问以大片连续访问为主，它们的页大小规划为 32K。这 6 个缓冲池都暂时规划为内存自动调整。

表 2-4　erpdb 数据库规划

数据库名字	erpdb	数据库用户	erpinst1
字符集	UTF-8	默认页大小	32K

缓冲池

缓冲池名	页大小	自动调整	描述
HR_DBP	8K	True	人力资源模块数据缓冲池
HR_IBP	8K	True	人力资源模块索引缓冲池
QRY_DBP	16K	True	分析查询模块数据缓冲池
QRY_IBP	16K	True	分析查询模块索引缓冲池
RPT_DBP	32K	True	报表模块数据缓冲池
RPT_IBP	32K	True	报表模块索引缓冲池

表空间（自动存储管理）

表空间	页大小	容器	描述
HR_DTS	8K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000005/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000005/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000005/C00000002.LRG	人力资源数据表空间
HR_ITS	8K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000006/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000006/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000006/C00000002.LRG	人力资源索引表空间
QRY_DTS	16K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000007/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000007/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000007/C00000002.LRG	分析查询数据表空间
QRY_ITS	16K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000008/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000008/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000008/C00000002.LRG	分析查询索引表空间
RPT_DTS	32K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000009/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000009/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000009/C00000002.LRG	报表数据表空间
RPT_ITS	32K	/db2fs1/erpinst1/NODE0000/ERPDB/T0000010/C0000000.LRG /db2fs2/erpinst1/NODE0000/ERPDB/T0000010/C00000001.LRG /db2fs3/erpinst1/NODE0000/ERPDB/T0000010/C00000002.LRG	报表索引表空间

规划好缓冲池后，就可以考虑表空间了，实际上表空间也是按照应用来确定的。从表 2-4 可以看出，人力资源模块包括数据表空间 HR_DTS 和索引表空间 HR_ITS，和所使用的缓冲池 HR_DBP、HR_IBP 的页大小一样，都是 8K；分析查询模块包括数据表空间 QRY_DTS 和索引表空间 QRY_ITS，和所使用的缓冲池 QRY_DBP、QRY_IBP 的页大小一样，都是 16K；报表模块包括数据表空间 RPT_DTS 和索引表空间 RPT_ITS，和所使用的缓冲池 RPT_DBP、RPT_IBP 的页大小一样，都是 32K。

为了方便维护和管理，选择了 DB2 V9.7 提供的表空间自动存储管理（Automatic Storage），这样只需要提供存储路径（Storage Path），具体的容器文件就可以由 DB2 自己来管理了，例如表空间 HR_DTS 的三个容器文件/db2fs1/erpinst1/NODE0000/ERPDB/T0000005/C0000000.LRG、/db2fs2/erpinst1/NODE0000/ERPDB/T0000005/C00000001.LRG 和 /db2fs3/erpinst1/NODE0000/ERPDB/T0000005/C00000002.LRG 都是由 DB2 来创建和管理的。

2. 创建 erpdb 数据库

从 DB2 V9.7 开始提供了 Oracle 兼容模式，以更好地帮助具有 Oracle 使用经验的 DBA 使用和维护 DB2。基于这个考虑，也为分析查询库打开 Oracle 兼容模式，即在环境变量中设定："DB2_COMPATIBILITY_VECTOR=ORA"，设置后重新启动实例。

具体的命令和系统返回信息如下：

```
=> db2set DB2_COMPATIBILITY_VECTOR=ORA
=> db2stop
SQL1063N  DB2STOP processing was successful.
=> db2start
SQL1063N  DB2START processing was successful.
```

随后，创建数据库 ERPDB。命令如下所示，其中数据库别名为 erpdb2，automatic storage 表示启用表空间的自动存储管理，codeset UTF-8 表示使用 UTF-8 编码，territory CN 表示区域为中国，pagesize 32 k 表示 erpdb 的默认页大小为 32K。

```
    => db2 create database erpdb automatic storage yes on /db2fs1, /db2fs2, /db2fs3 ALIAS
erpdb2 using codeset UTF-8 territory CN pagesize 32 k
    DB20000I  The CREATE DATABASE command completed successfully.
```

最后，连接到数据库 ERPDB 并为它设置如下参数，命令如下所示：

```
update db cfg using auto_reval DEFERRED_FORCE;
update db cfg using decflt_rounding ROUND_HALF_UP;
```

上面这两个参数的具体含义是什么？接下来我们就对这两个参数寻个究竟。

（1）参数 AUTO_REVAL：当遇到无效对象时，AUTO_REVAL 不同的设置决定了数据库对此无效对象不同的行为。AUTO_REVAL 的默认值是 DEFERRED，意思是如果一个对象例如视图或函数是无效的，下一次如果被引用将会自动尝试使之生效。在上面的设置中，我们将其从默认值调整为 DEFERRED_FORCE，这有什么好处呢？我们用一个例子来说明。例如，有一个存储过程依赖一个尚未创建完成的表，如果该参数设置为 DEFERRED_FORCE，那么该存储过程也能成功完成创建，但是，它将被标注为无效的。等到依赖表成功创建并且该存储过程第一次被调用时，DB2 将自动使该存储过程生效。这样带来的好处是非常明显的：有大量的对象需要创建，但是确定它们之间的依赖顺序非常难，将该参数设置为 DEFERRED_FORCE 将解决这一难题。

（2）参数 DECFLT_ROUNDING：如果应用程序中用到了浮点类型 DECFLOAT,那么必然会有舍入模式（rounding mode）的问题。DECFLT_ROUNDING 的默认设置是 round-half-even，它和 Oracle 的默认舍入模式不同。读者可以将该设置调整为 round-half-up，这样就和 Oracle 完

全匹配了，从而方便了应用程序的开发和迁移。

3. 创建缓冲池和表空间

数据库创建完成后，就可以根据表 2-4 的规划创建缓冲池和表空间了。

（1）创建缓冲池

使用下面的语句，分别创建了 HR_DBP、HR_IBP、QRY_DBP、QRY_IBP、RPT_DBP 和 RPT_IBP 缓冲池，其中 PAGESIZE 表示页面大小，创建完毕后，这些缓冲池大小是由 STMM 自动管理的。通常使用 STMM 自动调整就可以了，但是这个案例中由于 X3850 服务器的内存非常多，为了避免 STMM 带来的内存分配和回收的开销，所以在后文的 2.5.1 节还会将缓冲池大小手工调整为固定大小。

```
CREATE BUFFERPOOL "HR_DBP"  PAGESIZE 8192
CREATE BUFFERPOOL "HR_IBP"  PAGESIZE 8192
CREATE BUFFERPOOL "QRY_DBP" PAGESIZE 16384
CREATE BUFFERPOOL "QRY_IBP" PAGESIZE 16384
CREATE BUFFERPOOL "RPT_DBP" PAGESIZE 32768
CREATE BUFFERPOOL "RPT_IBP" PAGESIZE 32768
```

（2）创建表空间

使用下面的语句，分别创建了 HR_DTS、HR_ITS、QRY_DTS、QRY_ITS、RPT_DTS 和 RPT_ITS。由于在 create database 语句中启用了 Automatic Storage，所以创建表空间的语句就很简单了，不需要显示指定容器文件，DB2 可以自动完成创建和管理。

```
CREATE TABLESPACE HR_DTS BUFFERPOOL HR_DBP NO FILE SYSTEM CACHING
CREATE TABLESPACE HR_ITS BUFFERPOOL HR_IBP NO FILE SYSTEM CACHING
CREATE TABLESPACE QRY_DTS BUFFERPOOL QRY_DBP NO FILE SYSTEM CACHING
CREATE TABLESPACE QRY_ITS BUFFERPOOL QRY_IBP NO FILE SYSTEM CACHING
CREATE TABLESPACE RPT_DTS BUFFERPOOL RPT_DBP NO FILE SYSTEM CACHING
CREATE TABLESPACE RPT_ITS BUFFERPOOL RPT_IBP NO FILE SYSTEM CACHING
```

2.2.6　DB2 参数配置

DB2 的参数包括三套，即环境变量、数据库管理器配置参数（DBM CFG）和数据库配置参数（DBM CFG）。

注意：DB2 的参数设置

这里给出的只是初始化参数配置，在后面 2.5 节的优化实施中，将会根据具体情况再加以适当调整。

1. 设置 DB2 环境变量

环境变量的设置如下，其中“DB2COMM=TCPIP”是为了支持通过 TCP / IP 网络访问 DB2 数据库；“DB2CODEPAGE=1208” 指定了 UTF-8 编码，这样可以方便 db2move、import、export 等工具进行数据导入导出时使用正确的编码。

```
=> db2set DB2COMM=TCPIP
```

```
=> db2set DB2CODEPAGE=1208
```

2. 设置 DBM CFG

参数 DIAGLEVEL 的值设定为 3，表示在诊断日志文件中记录所有的错误和警告。关于所有设定值的含义，如下所示：

0 ——没有捕获到诊断数据

1 ——仅严重错误

2 ——所有错误

3 ——所有错误和警告

4 ——所有错误、警告以及参考消息

参数 DIAGSIZE 的值如果为 0，则表示不启动轮转模式，只会有一个诊断日志文件 db2diag.log。下面将其值设定为 1024，表示启用最多诊断日志文件个数为 10 的轮转模式，这些轮转诊断日志的最大值为 1024M，也就是说每个轮转日志文件最大为 100M（1024M /10 个轮转文件）。轮转模式将使用 10 个轮换诊断日志文件和 10 个轮换管理通知日志文件，这些文件名为 db2diag.n.log 和<instance>.n.nfy，其中，n 是整数。db2diag.n.log 文件和<instance>.n.nfy 文件的数目都不能超过 10。当第 10 个文件的大小达到限制后，将删除最旧的文件并重新创建新文件。

参数 SHEAPTHRES 表示整个实例范围内私有排序总内存的软限制，如果 sheapthres 被设置为 0，则 DB2 使用共享排序。如果 sheapthres 被设置为非 0，表示 DB2 将使用私有排序，排序内存在代理中分配，SORTHEAP 参数此时则规定了每个私有排序最多使用的内存量。这个案例中，设定 SHEAPTHRES 的值为 0，这时 DB2 使用共享排序。

最终，DBM CFG 的设置如下：

```
update dbm cfg using DIAGLEVEL 3
db2 update dbm cfg using DIAGSIZE 1024
update dbm cfg using SHEAPTHRES 0
```

3. 设置 DB CFG

DB CFG 的配置比较多，下面我们分几部分阐述一下。

（1）预取和清页进程数目设定。

参数 NUM_IOSERVERS 用于指定数据页预取的 I/O 服务器数目，NUM_IOCLEANERS 用于指定异步页清除代理程序的数目。为了方便，这两个参数都设定为 AUTOMATIC，这样就可以让 DB2 来自动管理了。

```
update db cfg using NUM_IOCLEANERS AUTOMATIC
update db cfg using NUM_IOSERVERS AUTOMATIC
```

（2）锁参数设定

LOCKLIST 设定为 40960 个 4K 页面，指定了分配给锁列表的存储容量。MAXLOCKS 指定了允许应用程序占用的锁列表的百分比，设定为 40，意味着当一个应用程序所使用的锁列表百分比达到 40%时，数据库管理器会升级其中的行锁，即用表锁代替行锁，以减少锁的数量。

LOCKTIMEOUT 指定应用程序为获取锁需等待的最多的秒数，设定为 30 秒。

```
update db cfg using LOCKLIST 40960
update db cfg using MAXLOCKS 40
update db cfg using LOCKTIMEOUT 30
```

（3）日志参数设定

LOGBUFSZ 表示日志缓冲区大小，设定为 10240，单位为 4K 页面；LOGFILSIZ 表示日志文件大小，设定为 102400，单位为 4K 页面，这样每个日志文件的大小为 400 MB；LOGPRIMARY 为主日志文件个数，设定为 50，LOGSECOND 为辅助日志文件个数，设定为 80，这样最多事务日志文件个数为 130 个，每个日志文件大小为 500M，所以可以支持最大日志量约为 50G（130*400M）左右的事务。

SOFTMAX 参数决定了 DB2 在灾难恢复时需要读取日志文件的数目，同时它也决定了数据缓冲区向磁盘写入数据的频率，把它设定为 150。当把 SOFTMAX 值设置较小时，则当发生灾难恢复时的恢复速度会变快，但是数据缓冲区写数据的频率会变高，即 I/O 次数会变大。反之，如果采用较大的 SOFTMAX 值，则数据库 I/O 频率会降低，但是一旦发生灾难恢复时，DB2 进行恢复的时间会延长。一般情况下，SOFTMAX 值设置为 100~300 比较合适。

NEWLOGPATH 设定为“/db2log/trans”，这是事务日志的存放路径，LOGARCHMETH1 设定为“DISK:/db2log/arch”，这是归档日志的存放路径，这也是按照当初的规划定义的。

最终，日志相关参数的配置如下：

```
update db cfg using LOGBUFSZ 10240
update db cfg using LOGFILSIZ 102400
update db cfg using LOGPRIMARY 50
update db cfg using LOGSECOND 80
update db cfg using SOFTMAX 150
update db cfg using TRACKMOD ON
update db cfg using NEWLOGPATH /db2log/trans
update db cfg using LOGARCHMETH1 DISK:/db2log/arch
```

（4）内存参数设定

内存参数的设定是最复杂的，这是由于这些参数的值需要根据负载来动态调整，如果靠 DBA 来手工调整是不现实的。不过，这可以让 DB2 来自动调整，例如下面的多个内存相关参数都设定为 AUTOMATIC，包括目录高速缓存大小（CATALOGCACHE_SZ）、程序包高速缓存大小（PCKCACHESZ）、语句堆大小（STMTHEAP）、应用程序堆大小（APPLHEAPSZ）、统计信息堆大小（STAT_HEAP_SZ）、共享排序的排序堆阈值（SHEAPTHRES_SHR）和排序堆大小（SORTHEAP），还有其他的一些内存参数，此处不再列出。

```
update db cfg using CATALOGCACHE_SZ AUTOMATIC
update db cfg using PCKCACHESZ AUTOMATIC
update db cfg using STMTHEAP AUTOMATIC
update db cfg using APPLHEAPSZ AUTOMATIC
update db cfg using STAT_HEAP_SZ AUTOMATIC
update db cfg using SHEAPTHRES_SHR AUTOMATIC
update db cfg using SORTHEAP AUTOMATIC
...
```

2.3 实施：DB2 HADR

在 2.1.3 小节规划了 DB2 HADR 高可用方案，包括 DB2 主节点和从节点，服务器都为性价比极高的 x3850 服务器，64 个内核、256GB 内存。为了真正满足灾备的需要，DB2 主节点和从节点是跨机房部署的，其中主节点部署在北京大兴机房，从节点部署在北京亦庄机房，两个节点之间通过专用私有网络连接，选择近同步工作模式。

当前，主节点的数据库尺寸已经达到了 8TB 左右，这个尺寸已经很大了，但是客户允许主节点的最多停机时间竟然只有 24 个小时！

这一节，重点讲述一下实施过程。

2.3.1 准备工作

在 DB2 HADR 的实施之前，需要预先准备好环境。

首先，主节点服务器和从节点服务器的操作系统版本必须完全相同。这个案例中，主节点和从节点操作系统的版本是完全一致性的，都是 Redhat 企业版 5.7。

其次，在从节点服务器上安装和主节点完全一致的 DB2 版本。参考 2.2.1~2.2.4 小节，在从节点上也执行和主节点一致的过程，包括 DB2 安装、操作系统参数配置、实例创建以及存储规划与文件系统创建。

> **注意：从节点上的 DB2 参数不需要手工配置**
>
> 不需要在从节点上创建数据库和进行参数配置工作，在后面的 2.3.2 小节，读者就可以理解，从节点上的 DB2 数据库文件从主节点直接复制过来，随后再编目（catalog）就可以了。

第三，对 HADR 主节点-从节点连接使用专用网络。应用通过千兆以太网络连接主节点数据库以及从节点数据库，为了不占用应用网络的带宽，在主节点和从节点内部通信上使用单独的网络，代码如下所示，其中“192.168.3.1”为主节点配置的 IP 地址，对应的域名为“PrimaryNode-2”，“192.168.3.2”为从节点配置的 IP 地址，对应的域名为“StandbyNode-2”。

```
[root@PrimaryNode-2 ~]# cat /etc/hosts
127.0.0.1             localhost.localdomain localhost
10.2.3.1          PrimaryNode-1
10.2.3.2          StandbyNode-1
192.168.3.1        PrimaryNode-2
192.168.3.2        StandbyNode-2
```

2.3.2 复制数据库

本小节讲到的复制数据库，意思是把主节点上的 erpdb 数据文件复制到从节点所在的服务器上，需要复制的数据库文件路径如下，其中/db2fs1、/db2fs2 和/db2fs3 为数据文件所在路径，/db2log/trans 为事务日志所在路径。

```
/db2fs1
/db2fs2
/db2fs3
/db2log/trans
```

注意：为什么不采用主节点备份和从节点恢复的方式？

配置 HADR，通常的做法是在主节点对数据库做备份，随后把备份介质从主节点复制到从节点，随后在从节点恢复。这种做法适合数据库尺寸较小的场景，否则备份、介质传输和恢复需要的时间过长，会影响业务使用。

在本案例中，数据库尺寸已经达到了 8TB 左右，而且主节点的最大允许停机时间只有 24 个小时，所以我们采用了将主节点的数据库文件复制到从节点，随后在从节点 catalog 的方式，这样就节省了备份和恢复的时间。

接下来，我们详细讲解一下实施步骤。

（1）对主节点数据库停机

在主节点上，执行下面的命令：

```
[erpinst1@PrimaryNode-2 ~]$ db2 force applications all
[erpinst1@PrimaryNode-2 ~]$ db2 deactivate db ERPDB
```

（2）暂挂主节点数据库上的 I/O

在主节点上，执行下面的命令：

```
 [erpinst1@PrimaryNode-2 ~]$ db2 set write suspend for database
DB20000I  The SET WRITE command completed successfully.
```

（3）复制主节点的数据文件和日志文件

在这个案例中，我们选择了 rsysn 工具实现数据文件的复制工作，具体步骤如下：

注意：为什么选择 rsync 工具？

rsync（Remote Synchronize）是 Unix/Linux 系统下的数据镜像备份工具，它基于客户端-服务器端工作模式，支持断点续传，是一款快速实现远程同步功能的软件，在同步目录和文件的同时，可以保持原来文件的权限、时间、软硬链接等附加信息。

本案例中，在从主节点向从节点复制数据文件时，需要确保文件和路径完全一致，而且中间传输中出现问题时，还希望实现断点续传，这些要求正是 rsync 工具最擅长做的，所以最终选择了 rsync 工具。

① 编辑配置文件

在这个案例中，主节点 PrimaryNode-2 为 rsync 的服务器端，从节点 StandbyNode-2 为 rysnc 的客户端，在主节点上编辑/etc/rsyncd.conf 文件，配置需要复制的目录，如下所示：

```
[root@PrimaryNode-2~]# cat /etc/rsyncd.conf
[db2fs1]
path = /db2fs1/
auth users = erpinst1
uid = root
```

```
gid = root
secrets file = /etc/rsyncd.secrets
read only = no

[db2fs2]
path = /db2fs2/
auth users = erpinst1
uid = root
gid = root
secrets file = /etc/rsyncd.secrets
read only = no

[db2fs3]
path = /db2fs3/
auth users = erpinst1
uid = root
gid = root
secrets file = /etc/rsyncd.secrets
read only = no

[db2log]
path = /db2log/trans
auth users = erpinst1
uid = root
gid = root
secrets file = /etc/rsyncd.secrets
read only = no
```

在/etc/rsyncd.conf 文件中，应用了密码文件/etc/rsyncd.secrets，这个文件中的用户名和密码是 rsync 客户端复制服务器端数据时需要提供的。这个文件的内容如下所示，用户名为“erpinst1”，密码为“password”。

```
[root@PrimaryNode-2 ~]# cat  /etc/rsyncd.secrets
erpinst1:password
```

② 启动 rsync 服务

首先，编辑/etc/xinetd.d/rsync 文件，将“disable = yes”改为“disable = no”，这表示在主节点上启用 rsync 服务。

```
[root@PrimaryNode-2 ~]# cat /etc/xinetd.d/rsync
# default: off
# description: The rsync server is a good addition to an ftp server, as it \
#      allows crc checksumming etc.
service rsync
{
#需要从 yes 修改为 no
       disable = no
       socket_type     = stream
       wait            = no
       user            = root
       server          = /usr/bin/rsync
       server_args     = --daemon
```

```
        log_on_failure  += USERID
}
```

随后，如下所示，在主节点上启动 xinetd 服务：

```
[root@PrimaryNode-2 ~]# service xinetd restart
Stopping xinetd: [  OK  ]
Starting xinetd: [  OK  ]
```

③ 开始同步

为了提高效率，可以同时使用主节点上的两块网卡，即外网 IP 地址 10.2.3.1 和内网 IP 地址 192.168.3.1，同时对数据文件路径和日志文件路径进行同步。

复制 db2fs1 目录，使用 PrimaryNode-1 域名，即 IP 地址为 10.2.3.1，命令如下：

```
[root@StandbyNode-2 ~]# cat rsync1.sh
DATE=`date +%Y%m%d%H%M`
nohup rsync -aSvH --password-file /etc/rsyncd.secrets erpinst1@ PrimaryNode-1::db2fs1
/db2fs1 >/var/log/rsync.data1.$DATE &
[root@StandbyNode-2 ~]# ./rsync1.sh
```

复制 db2fs2 目录，使用 PrimaryNode-1 域名，即 IP 地址为 10.2.3.1，命令如下：

```
[root@StandbyNode-2 ~]# cat rsync2.sh
DATE=`date +%Y%m%d%H%M`
nohup rsync -aSvH --password-file /etc/rsyncd.secrets erpinst1@ PrimaryNode-1::db2fs2
/db2fs2 >/var/log/rsync.data2.$DATE &
[root@StandbyNode-2 ~]# ./rsync2.sh
```

复制 db2fs3 目录，使用 PrimaryNode-2 域名，即 IP 地址为 192.168.3.1，命令如下：

```
[root@StandbyNode-2 ~]# cat rsync3.sh
DATE=`date +%Y%m%d%H%M`
nohup rsync -aSvH --password-file /etc/rsyncd.secrets erpinst1@ PrimaryNode-2::db2fs3
/db2fs3 >/var/log/rsync.data3.$DATE &
[root@StandbyNode-2 ~]# ./rsync3.sh
```

复制 db2log 目录，使用 PrimaryNode-2 域名，即 IP 地址为 192.168.3.1，命令如下：

```
[root@StandbyNode-2 ~]# cat rsynclog.sh
DATE=`date +%Y%m%d%H%M`
nohup rsync -aSvH --password-file /etc/rsyncd.secrets erpinst1@ PrimaryNode-2::db2log
/db2log/trans >/var/log/rsync.trans.$DATE &
[root@StandbyNode-2 ~]# ./rsynclog.sh
```

④ 监控复制过程

由于需要复制的文件比较大，有 8TB 左右，通过千兆以太网复制的话，至少需要 10 小时左右，期间需要监控进程和网络情况。

例如，使用“ps -ef”命令监控 rsync 进程，可以看到 4 个进程，其中 3 个是复制数据目录的，1 个是复制事务日志目录的。

```
[root@StandbyNode-2 ~]# ps -ef | grep rsync
root      7011  6592  0 21:34 pts/1    00:00:00 grep rsync
root     24170 24169 50 18:24 ?        01:35:55 rsync -aSvH --password-file
```

```
/etc/rsyncd.secrets erpinst1@PrimaryNode-1::db2fs1 /db2fs1
    root     24340 24339 51 18:25 ?       01:37:59 rsync -aSvH --password-file
/etc/rsyncd.secrets erpinst1@PrimaryNode-1::db2fs2 /db2fs2
    root     24370 24369 83 18:26 ?       02:37:42 rsync -aSvH --password-file
/etc/rsyncd.secrets erpinst1@PrimaryNode-2::db2fs3 /db2fs3
    root     24370 24399 83 18:26 ?       02:37:42 rsync -aSvH --password-file
/etc/rsyncd.secrets erpinst1@PrimaryNode-2::db2log /db2log/trans
```

例如，使用 nmon 命令监控网络情况，其中 eth0 为连接外网的网卡，IP 地址为 10.2.3.1，eth4 为连接内网的网卡，IP 地址为 192.168.3.1。从下面的输出可以看到，两个网卡都同时在工作。

```
     [root@PrimaryNode-2 ~]# nmon

     ┌nmon─14g───────────────────────────────Hostname=db2serv2───────Refresh=
2secs ────21:36.35──────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────┐
     | Network I/O ─────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────        |
     |I/F Name Recv=KB/s Trans=KB/s packin packout insize outsize Peak->Recv Trans |
     |    lo    26.2    26.2     536.0    536.0    50.0   50.0      52.2    52.2   |
     |   eth4 38430.6   176.2    26297.0   2573.8  1496.5   70.1     84480.6   458.0   |
     |   eth5     0.0     0.0       0.0      0.0     0.0    0.0       0.0    0.0   |
     |   eth6     0.0     0.0       0.0      0.0     0.0    0.0       0.0    0.0   |
     |   usb0     0.1     0.0       1.0      0.5    65.0   42.0       0.1     0.0  |
     |   eth7     0.0     0.0       0.0      0.0     0.0    0.0       0.0    0.0   |
     |   usb1     0.0     0.0       0.0      0.0     0.0    0.0      0.1    0.0    |
     |   eth0 103935.2   723.4    70585.8   11221.8  1507.8   66.0    182302.1  1382.9 |
     |   eth1     0.0     0.0       0.0      0.0     0.0    0.0       0.0     0.0  |
     |   eth2     0.0     0.0       0.0      0.0     0.0    0.0       0.0     0.0  |
     |   eth3     0.0     0.0       0.0      0.0     0.0    0.0       0.0     0.0  |
```

⑤ 完成后检查

执行完 rsync 同步后，使用 find 命令检查主节点和从节点的/db2fs1、/db2fs2、/db2fs3 和/db2log/trans 下的文件个数及大小是否一致。

```
    find /db2fs1 -type f -ls | wc -l
    find /db2fs2 -type f -ls | wc -l
    find /db2fs2 -type f -ls | wc -l
    find /db2log/trans -type f -ls | wc -l
```

确认完全一致后，就说明 rsync 已经成功把数据和日志从主节点复制到从节点上了。

（4）恢复主节点数据库上的 I/O

使用下面的命令进行恢复，这样主数据库就可以继续接受访问了。

```
    [erpinst1@PrimaryNode-2 ~]$ db2 set write resume for database
    DB20000I  The SET WRITE command completed successfully.
```

（5）对从节点数据库进行编目

如下所示，这可以使用 catalog database 命令来完成：

```
[erpinst1@StandbyNode-2 ~]$ db2 catalog database erpdb on /db2fs1
DB20000I  The CATALOG DATABASE command completed successfully.
DB21056W  Directory changes may not be effective until the directory cache is
refreshed.
```

（6）对从节点数据库进行初始化

如下所示，首先使用 db2start 命令在从节点上启动数据库，随后使用“db2initdb erpdb as standby”来初始化从节点数据库。

```
 [erpinst1@StandbyNode-2 ~]$ db2start
11/30/2012 22:54:23     0   0   SQL1063N  DB2START processing was successful.
SQL1063N  DB2START processing was successful.

[erpinst1@StandbyNode-2 ~]$ db2inidb erpdb as standby
DBT1000I  The tool completed successfully.
```

2.3.3 配置 HADR

在正式启动 HADR 主节点和从节点之前，需要进行必要的配置工作，具体包括下面一些配置步骤。

（1）配置 HADR 的通讯端口

在主节点和从节点的/etc/services 文件末尾，加入一行“db2h_db2inst170000/tcp”，用以指定主节点和从节点之间的通讯端口。

```
db2c_erpinst1  50001/tcp # DB2 connection service port
DB2_db2inst1    60000/tcp
DB2_db2inst1_1  60001/tcp
DB2_db2inst1_2  60002/tcp
DB2_db2inst1_END        60003/tcp
db2h_erpinst1   70000/tcp #db2 hadr service port
```

（2）数据库参数配置

主节点的 DB CFG 配置：

```
 [erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using hadr_local_host PrimaryNode-2
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using hadr_local_svc db2h_erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_host StandbyNode-2
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_svc db2h_erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_inst erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using logindexbuild on
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg for erpdb using indexrec restart
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

从节点的 DB CFG 配置：

```
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using hadr_local_host StandbyNode-2
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using hadr_local_svc db2h_erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_host PrimaryNode-2
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_svc db2h_erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using hadr_remote_inst erpinst1
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using logindexbuild on
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@StandbyNode-2 ~]$ db2 update db cfg for erpdb using indexrec restart
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

（3）注册变量配置

DB2 从 V9.7 开始增加了从节点读特性，即 Read On Standby（RoS），另外，从节点读只支持 UR（Uncommit Read）隔离级别。所以，可以通过 db2set 命令，在主节点和从节点同时设置“DB2_HADR_ROS=ON”和“DB2_STANDBY_ISO=UR”，如下所示：

主节点设置：

```
[erpinst1@PrimaryNode-2 ~]$ db2set DB2_HADR_ROS=ON
[erpinst1@PrimaryNode-2 ~]$ db2set DB2_STANDBY_ISO=UR
[erpinst1@PrimaryNode-2 ~]$ db2set
DB2_COMPATIBILITY_VECTOR=ORA
DB2COMM=TCPIP
DB2CODEPAGE=1208
DB2_HADR_ROS=ON
DB2_STANDBY_ISO=UR
```

从节点设置：

```
[erpinst1@StandbyNode-2 ~]$ db2set DB2_HADR_ROS=ON
[erpinst1@StandbyNode-2 ~]$ db2set DB2_STANDBY_ISO=UR
[erpinst1@StandbyNode-2 ~]$ db2set
DB2_COMPATIBILITY_VECTOR=ORA
DB2COMM=TCPIP
DB2CODEPAGE=1208
DB2_HADR_ROS=ON
DB2_STANDBY_ISO=UR
```

2.3.4 启动主从数据库

启动时，按照先从节点后主节点的顺序启动。

（1）启动从节点数据库

如下所示，首先使用 db2start 命令在从节点上启动数据库，随后使用“START HADR ON DB ERPDB AS STANDBY”命令启动从节点数据库。

```
 [erpinst1@StandbyNode-2 ~]$ db2start
11/30/2012 22:54:23     0   0   SQL1063N  DB2START processing was successful.
SQL1063N  DB2START processing was successful.

[erpinst1@StandbyNode-2 ~]$ db2 START HADR ON DB ERPDB AS STANDBY
DB20000I  The START HADR ON DATABASE command completed successfully.
```

（2）启动主节点数据库

如下所示，首先使用 db2start 命令在主节点上启动数据库，随后使用“START HADR ON DB ERPDB AS PRIMARY”来初始化主节点数据库。

```
 [erpinst1@PrimaryNode-2 ~]$ db2start
11/30/2012 22:54:23     0   0   SQL1063N  DB2START processing was successful.
SQL1063N  DB2START processing was successful.

[erpinst1@PrimaryNode-2 ~]$ db2 START HADR ON DB ERPDB AS PRIMARY
DB20000I  The START HADR ON DATABASE command completed successfully.
```

2.3.5 HADR 切换演练

在主节点发生故障时，从节点可以自动或者手动接管主节点，下面先进行一下手工切换演练，这样当真正出现故障时就不会慌乱了。

1. 切换前状态

使用 db2pd 监控主节点数据库（PriamryNode-2 这时是主节点）：

```
 [erpinst1@PriamryNode-2 ~]$ db2pd -d erpdb -hadr

Database Partition 0 -- Database ERPDB -- Active -- Up 0 days 00:16:11 -- Date 12/01/2012
00:09:20

HADR Information:
Role    State              SyncMode   HeartBeatsMissed   LogGapRunAvg (bytes)
Primary Peer               Async    0                 0

ConnectStatus ConnectTime                          Timeout
Connected     Sat Dec  1 00:08:05 2012 (1354291685) 120

LocalHost                               LocalService
PrimaryNode-2                               db2h_erpinst1

RemoteHost                              RemoteService     RemoteInstance
StandbyNode-2                               db2h_erpinst1      erpinst1

PrimaryFile  PrimaryPg  PrimaryLSN
S0000002.LOG 6         0x00001F2FE832EE75

StandByFile  StandByPg  StandByLSN
S0000002.LOG 0         0x00001F2FE8328010
```

使用 db2pd 监控从节点数据库（StandbyNode-2 这时是从节点）：

```
 [erpinst1@StandbyNode-2 ~]$ db2pd -d erpdb -hadr

Database Partition 0 -- Database ERPDB -- Active Standby -- Up 0 days 00:03:28 -- Date
12/01/2012 00:07:57

HADR Information:
Role    State              SyncMode   HeartBeatsMissed   LogGapRunAvg (bytes)
Standby Peer               Async    0                  0

ConnectStatus ConnectTime                           Timeout
Connected     Sat Dec  1 00:07:13 2012 (1354291633) 120

ReplayOnlyWindowStatus ReplayOnlyWindowStartTime           MaintenanceTxCount
Inactive               N/A                                 0

LocalHost                                 LocalService
StandbyNode-2                               db2h_erpinst1

RemoteHost                                RemoteService     RemoteInstance
PrimaryNode-2                               db2h_erpinst1      erpinst1

PrimaryFile  PrimaryPg  PrimaryLSN
S0000002.LOG 0          0x00001F2FE8328010

StandByFile  StandByPg  StandByLSN         StandByRcvBufUsed
S0000002.LOG 0          0x00001F2FE8328010 0%
```

2. 切换

在从节点上执行“takeover hadr”命令，这样从节点将接管主节点，这是当出现异常情况后手工切换模式。

```
 [erpinst1@StandbyNode-2 ~]$ db2 takeover hadr on database erpdb
DB20000I  The TAKEOVER HADR ON DATABASE command completed successfully.
```

3. 切换后状态

使用 db2pd 命令监控主节点数据库（StandbyNode-2 这时变为主节点）：

```
 [erpinst1@StandbyNode-2 ~]$ db2pd -d erpdb -hadr

Database Partition 0 -- Database ERPDB -- Active -- Up 0 days 09:38:42 -- Date 12/01/2012
09:43:11

HADR Information:
Role    State              SyncMode   HeartBeatsMissed   LogGapRunAvg (bytes)
Primary Peer               Async    0                  0

ConnectStatus ConnectTime                           Timeout
Connected     Sat Dec  1 00:07:13 2012 (1354291633) 120

LocalHost                                 LocalService
StandbyNode-2                               db2h_erpinst1
```

```
RemoteHost                              RemoteService     RemoteInstance
PrimaryNode-2                              db2h_erpinst1      erpinst1

PrimaryFile  PrimaryPg  PrimaryLSN
S0000002.LOG 4920       0x00001F2FE966031A

StandByFile  StandByPg  StandByLSN
S0000002.LOG 4920       0x00001F2FE966031A
```

使用 db2pd 监控从节点数据库（PriamryNode-2 这时变为从节点）：

```
[erpinst1@PrimaryNode-2 ~]$ db2pd -d erpdb -hadr

Database Partition 0 -- Database ERPDB -- Standby -- Up 0 days 09:51:10 -- Date 12/01/2012
09:44:19

HADR Information:
Role    State              SyncMode   HeartBeatsMissed   LogGapRunAvg (bytes)
Standby Peer               Async    0                15041024

ConnectStatus ConnectTime                          Timeout
Connected     Sat Dec  1 00:08:05 2012 (1354291685) 120

LocalHost                              LocalService
PrimaryNode-2                              db2h_erpinst1

RemoteHost                              RemoteService     RemoteInstance
StandbyNode-2                              db2h_erpinst1      erpinst1

PrimaryFile  PrimaryPg  PrimaryLSN
S0000002.LOG 4920       0x00001F2FE966031A

StandByFile  StandByPg  StandByLSN        StandByRcvBufUsed
S0000002.LOG 0          0x00001F2FE8328010  0%
```

2.3.6　为 Java 应用配置 ACR

当主节点和从节点之间发生故障切换时，客户端应用程序的数据库连接可以通过自动客户端重新路由（ACR）机制转移到新的主节点，这需要在客户端配置 ACR 来实现。

DB2 分析库的应用使用 Java 语言开发的，下面将讲述在 Java 应用中，如何通过对 DataSource 和 JNDI 的设置来配置 ACR。

1. DataSoure 设置

下面的代码是 DataSoure 的例子，其中 ds 为 DataSource 属性对象。调用“setServerName”方法指定主节点域名为“PrimaryNode-1”，调用“setPortNumber”方法指定主节点的端口为 50001；调用“setClientRerouteAlternateServerName”方法指定从节点域名为“StandbyNode-1”，调用“setClientRerouteAlternatePortNumber”方法指定从节点端口为 50001。其中，“PrimaryNode-1”和“StandbyNode-1”都是在/etc/hosts 文件中指定的（见 2.3.1 节）。

```
ds.setDriverType(4);
ds.setServerName("PrimaryNode-1");
ds.setPortNumber("50001");
ds.setClientRerouteAlternateServerName("StandbyNode-1");
ds.setClientRerouteAlternatePortNumber("50001");
```

2. JNDI 设置

下面的代码是配置 JNDI 的例子，创建一个 DB2ClientRerouteServerList 的对象 address，随后调用 setPrimaryServerName 和 setPrimaryPortNumber 分别指定主节点域名“PrimaryNode-1”和端口号 50001；再调用 setAlternateServerName 和 setAlternatePortNumber 分别指定从节点域名“StandbyNode-1”和端口号 50001。

```
ds.setDriverType(4);
// Create a starting context for naming operations
InitialContext registry = new InitialContext();
// Create a DB2ClientRerouteServerList object
DB2ClientRerouteServerList address = new DB2ClientRerouteServerList();
// Set the server name and port number for the primary server
address.setPrimaryServerName("PrimaryNode-1");
address.setPrimaryPortNumber(50001);
// Set the  port number and server name for the alternate server
int[] port = {50001};
String[] server = {"StandbyNode-1"};
address.setAlternateServerName(server);
address.setAlternatePortNumber(port);

registry.rebind("serverList", address);
// Assign the JNDI name of the DB2ClientRerouteServerList object to the
// clientRerouteServerListJNDIName property
datasource.setClientRerouteServerListJNDIName("serverList");
```

2.4　性能分析

按照上面的步骤完成了 DB2 经营分析库的搭建，完成了 HADR 方案的实施并成功上线，从效果来看，取得了非常好的回报：分析查询、报表和人力资源管理等从原来的 Oracle RAC 数据库中剥离出来，在 DB2 中运行，查询时间满足了设计需求；改造后 Oracle RAC 数据库只运行供应链、生产、财务、营销等业务模块，所能处理的事务吞吐量达到了要求，相应的服务器负载也下降了 50%左右。这些获得了该汽车制造商极高的评价和认可。

由于前期的软硬件规划比较好，实施也比较专业，所以这个 DB2 经营分析库上线初期运行得非常好，查询的响应时间、事务吞吐量都能满足需要，但是运行了几个月后，开始出现了一些性能问题，这些问题中有的只是影响了应用，例如某些查询，有的影响了整个数据库，例如大量的锁等待、频繁的大表扫描、大量的排序溢出等。

从第 1 章了解到，性能优化的背后实际上是业务需求驱动的。首先需要从业务角度将问题分类并识别结果，即从业务角度确定问题，设定每个问题的优化目标，并设定每个问题的优先

级。随后从系统角度确定原因，弄清楚时间花在何处？时间是如何被消耗的？如何减少时间耗费？同时要考虑到不同问题之间的连带效应，避免解决一个问题的同时带来新的问题。

接下来，针对这个案例，我们就从业务分析开始，随后谈谈系统分析，最后给出优化策略和实施计划。

2.4.1 业务分析

提升数据库的性能其实是为了给应用带来价值，那么当开始处理性能问题的时候，必须先理解业务需求。

运行几个月后，出现的性能问题从应用角度看，是这样的：分析查询中一些根据动态条件进行查询的语句执行速度越来越慢，特别是忙时更严重；报表生成时间也随着数据量的增长而越来越长；人力资源管理模块中，当访问量很大，系统处于忙时事务响应时间显著提高。如表 2-5 所示。

表 2-5 性能问题（业务角度）

应用	问题描述	优化目标	发生时机
人力资源管理	事务响应时间越来越长，从刚上线的 30s 左右增加到 2 分钟以上，已经影响了用户的使用	要求响应时间稳定在 1 分钟以内	仅在系统忙时发生
查询分析	执行动态查询上存在性能问题，有的需要 5 分钟以上的时间才能获得访问结果，而前端的应用逻辑是 2 分钟数据库没有返回结果就认为是超时了	优化动态查询语句，在 2 分钟以内出结果	忙时和闲时都有发生，忙时更严重
报表	刚上线时报表生成时间为 8 个小时左右，目前已经达到了 20 个小时左右，随着数据量的不断增长，这个时间还在逐渐增加	寻找一种不完全依赖于数据量的方案，将报表的生成时间控制在维护窗口可接受的范围，即 8 小时左右	闲时和忙时都发生

2.4.2 系统分析

就性能优化实践来说，完成业务分析后接下来需要弄清楚整个系统的情况，这包括存储、主机、操作系统以及数据库服务器。这要和系统管理员以及网络管理员进行沟通，尽量全面地收集信息以准确定位问题。

考虑到篇幅所限，本章将以数据库为重点进行讲解。那么首先就需要监控，从而分析和定位问题。监控包括两个层面，第一层面是对数据库本身监控，分为忙时和闲时；第二个层面是对应用监控，包括分析查询、报表和人力资源管理。

那么怎么选择监控手段呢？DB2 提供了多种多样的监控手段，主要包括快照监视器（Snapshot Monitor）、事件监视器（Event Monitor）、监视器表函数（Monitoring Table Functions）、管理视图（Administrative Views）、db2pd 以及 db2top 等。

需要注意的是，在执行上面的监控命令前，首先需要打开监控开关，本书第 6 章将讲述如

何打开。打开开关后，可以使用 get monitor switches 命令查询各个快照开关的状态，如下所示，所有监控开关已经打开了：

```
=> db2 get monitor switches

            Monitor Recording Switches

Switch list for db partition number 0
Buffer Pool Activity Information  (BUFFERPOOL) = ON
Lock Information                       (LOCK) = ON
Sorting Information                    (SORT) = ON
SQL Statement Information         (STATEMENT) = ON
Table Activity Information            (TABLE) = ON  2012-11-07 15:33:10.888675
Take Timestamp Information        (TIMESTAMP) = ON  2012-11-07 15:20:45.884026
Unit of Work Information                (UOW) = ON
```

在性能分析时，我们将结合使用快照监视器、管理视图和 db2pd 等监控工具。根据性能优化实战经验，将特别侧重于下面几个方面：

- 缓冲池：包括缓冲池大小以及命中率等。
- 锁：锁内存的分配情况、锁等待发生次数、死锁发生次数等。
- 排序：排序内存的分配情况、排序溢出等情况。
- 表：表的冷热情况，表的大小等，特别是对于热小表、热大表、冷大表要有不同的处理办法。
- 分区：合理使用分区技术，例如表分区和 MDC 分区等，对数据库的性能影响很大。
- 索引：关注执行查询时索引的使用情况，是否存在没用的索引等。
- SQL 语句：重点关注数据库运行一段时间内，包括忙时和闲时，特别是执行时间最多、执行次数最多的前十位 SQL 语句。这些语句后续可以从索引角度进行优化，从锁的角度进行调整。
- 压缩：针对巨大表，关注压缩对性能的影响。
- 临时表：关注临时表的使用情况。
- 运维：重点关注表的统计信息是否最新，不是的话要及时运行 runstats，关注表是否需要重组，对需要重组的表及时运行 reorg。

2.4.3 优化方法学与计划

考虑到目前系统的情况，自下而上方法学是比较适合的，毕竟我们没有参与过包括人力资源、查询分析等业务系统的设计和开发过程，并不深入了解应用逻辑。不过幸运的是，我们可以在优化过程中得到数据库管理员和应用开发人员的技术支持。

针对这个优化项目，核心内容有两个：其一，确定优化顺序，先集中精力解决业务问题，还是从全局调整开始？其二，优化顺序确定后，有哪些手段可以高效地解决这些问题？其三，一个具体的实施计划是什么样的？

首先，先确定优化顺序。通常在做性能优化时，顺序上是有讲究的，不能一概而论。有的优化项目，实施周期很短而且性能非常严重，客户要求务必先解决业务问题，所以只能从业务

优化入手，数据库全局调整后续再做；有的优化项目，业务性能有问题，但是客户给的实施周期较长，所以可以先对数据库进行全局调整，随后再对业务性能各个击破。这个案例将首先全局调整，随后再对各个业务系统优化。

其次，谈谈手段。首先要进行“诊断”，要找到性能瓶颈，确定时间是如何被消耗的，例如多少时间花在了 CPU 上、多少时间花在了 I/O 上，问题产生的时间和环境、问题的症状等；其次是“开处方”，即提出解决问题的最可行方案。

最后，就可以正式制定实施计划了。如表 2-6 所示，首先对数据库进行全局调整和优化，随后并行开展人力资源管理和查询分析的优化工作，接下来是报表。另外，表 2-6 还列出了系统割接和运维支持，这在实战中应用很多，当然这不是本书重点，后文不再赘述。

表 2-6　优化时间表

项目	开始时间	结束时间	持续时间
数据库全局调整和优化	2010-12-1	2010-12-4	4 天
人力资源管理	2010-12-5	2010-12-7	3 天
查询分析	2010-12-8	2010-12-12	5 天
报表	2010-12-13	2010-12-15	3 天
系统割接	2010-12-16	2010-12-18	3 天
运维支撑	2010-12-19	2010-12-23	5 天

2.5　性能优化

本节将讲述这个案例的详细优化过程。这是一个真实的项目，我们在每次调整前都要准备优化报告，等客户签字确认后，先在测试环境验证，随后在客户 DBA 的配合下正式在生产库上实施。

2.5.1　全局调整和优化

在全局调整方面，重点是存储规划和内存参数调整。除此以外，本节还讲述了在实战中应用十分广泛的表调整，即根据表的冷热程度来灵活定义表的存放位置，从而提升 I/O 吞吐性能。

1. 存储规划和调整

单个硬盘的 I/O 吞吐量是有上限的，为了提升吞吐量，可以把多个硬盘放在一起，组成磁盘阵列（RAID）来满足大规模读写请求。通常在一个 OLTP 应用中，磁盘阵列的写速度至少要达到 200 M/s 以上，这可以通过 dd 命令来测试，如下所示：

```
dd if=/dev/zero of=/data1/test.file bs=8192 count=5000000
5000000+0 records in
5000000+0 records out
40960000000 bytes (41 GB) copied, 179.411 seconds, 228 MB/s
```

关于存储的规划和调整，请读者参考本书第 3 章。

2. 内存参数调整

考虑到硬盘读写速度的限制，DB2 需要用内存来缓存磁盘数据，从而提高读写速度。在性能优化实践中，需要重点调整的内存参数包括缓冲池、排序堆、包缓存以及编目缓存等。

（1）缓冲池配置

对数据库来说，正确的内存配置非常重要，首先是监控缓冲池。使用“db2pd –db erpdb -bufferpools”命令，观察缓冲池的命中率。下面是“db2pd –db erpdb -bufferpools”命令的输出：

```
Database Partition 0 -- Database ERPDB -- Active -- Up 5 days 23:19:55 -- Date 2013-01-27
14:44:16

Bufferpools:
First Active Pool ID      1
Max Bufferpool ID         7
Max Bufferpool ID on Disk 7
Num Bufferpools          11

Address           Id   Name            PageSz     PA-NumPgs  BA-NumPgs  BlkSize
NumTbsp    PgsToRemov CurrentSz  PostAlter  SuspndTSCt Automatic
0x000000000B2F93A0 1    IBMDEFAULTBP     8192       1000       0         0         5
0         1000       1000       0         True
0x000000000B2F9840 2    HR_DBP          8192       12000      0         0         1
0         12000      12000      0         True
0x000000000B2F9CE0 3    HR_IBP          8192       4000     0         0         1         0
4000     4000     0         True
0x000000000B2FA180 4    QRY_DBP         16384      6000      0         0         1
0         6000      6000      0         True
0x000000000B2FA620 5    QRY_IBP         16384      4000      0         0         1
0         4000      4000      0         True
0x000000000B2FAAC0 6    RPT_DBP         32768      8000      0         0         1
0         8000      8000      0         True
0x000000000B2FAF60 7    RPT_IBP         32768      6000     0         0         1
0          6000      6000     0         True
0x000000000B2F8120 4096 IBMSYSTEMBP4K     4096      16        0         0         0
0         16        16        0         False
0x000000000B2F85C0 4097 IBMSYSTEMBP8K     8192      16        0         0         0
0         16        16        0         False
0x000000000B2F8A60 4098 IBMSYSTEMBP16K    16384     16        0         0         0
0         16        16        0         False
0x000000000B2F8F00 4099 IBMSYSTEMBP32K    32768     16        0         0         0
0         16        16        0         False

Bufferpool Statistics for all bufferpools (when BUFFERPOOL monitor switch is ON):

BPID DatLRds    DatPRds    HitRatio TmpDatLRds TmpDatPRds HitRatio IdxLRds    IdxPRds
HitRatio TmpIdxLRds TmpIdxPRds HitRatio
1    142234263  13827622   90.28%   14220952   809871     94.31%   127553113  1271518
99.00%   1736       0         100.00%
2    173        105        39.31%  0          0         00.00%   20         6         70.00%
```

```
0         0         00.00%
3    25535801   19642627   23.08%   0          0          00.00%   0          0
00.00%   0          0          00.00%
4    5848135    1402600       76.01%   0          0          00.00%   0          0
00.00%   0          0          00.00%
5    3673212    1471370      59.94%   0          0          00.00%   0          0
00.00%   0          0          00.00%
6    4799093        1309121        72.72%   0          0          00.00%   3108461    40819
98.69%   0          0          00.00%
7    511513       233123         54.42%   0          0          00.00%   28300983   174
100.00%   0          0          00.00%
4096 0          0          00.00%   0          0          00.00%   0          0          00.00%
0         0         00.00%
4097 0          0          00.00%   0          0          00.00%   0          0          00.00%
0         0         00.00%
4098 0          0          00.00%   0          0          00.00%   0          0          00.00%
0         0         00.00%
4099 0          0          00.00%   0          0          00.00%   0          0          00.00%
0         0         00.00%
```

从上面输出中可以看到各个缓冲池的页面分配情况，如表 2-7 所示，每个缓冲池分配的页面并不多，命中率也比较低。

表 2-7　缓冲池页面分配情况和命中率

缓冲池	分配页面个数	命中率	Automatic
HR_DBP	12000	39.31%	是
HR_IBP	4000	23.08%	是
QRY_DBP	6000	76.01%	是
QRY_IBP	4000	59.94%	是
RPT_DBP	8000	72.72%	是
RPT_IBP	6000	54.42%	是

小技巧：使用固定大小的缓冲池

运行 DB2 经营分析库的 x3850 服务器有 256GB 内存，针对当前应用来说足够多了，但是假如开启内存自调优的话，DB2 需要根据负载的情况自动为缓冲池分配或者回收内存，这对内存足够多的系统来说其实未必实用。我们可以自己为每个缓冲池配置固定大小的内存，避免 DB2 频繁动态分配或者回收内存。

首先，使用下面的命令查看内存自调优（STMM）是否打开：

```
[erpinst1@PrimaryNode-2 ~]$ db2 get db cfg for erpdb| grep -i 'self'
自调整内存                (SELF_TUNING_MEM) = ON
```

现在，我们为每个缓冲池的内存大小配置为常量，从而避免内存自动调优的开销以提高性能。经过监控发现，当前系统可用内存为 150GB 左右，将其中的 50%即大约 70GB 固定分配给 DB2 缓冲池使用，如表 2-8 所示，HR_DBP、QRY_DBP 和 RPT_DBP 规划为 16 GB，HR_IBP、

QRY_IBP 和 RPT_IBP 规划为 8 GB，总的缓冲池大小为 72 GB。

表 2-8 缓冲池尺寸规划

缓冲池（非自动调整）			
缓冲池名	页大小	尺寸	描述
HR_DBP	8K	16 GB	人力资源模块数据缓冲池
HR_IBP	8K	8 GB	人力资源模块索引缓冲池
QRY_DBP	16K	16 GB	分析查询模块数据缓冲池
QRY_IBP	16K	8 GB	分析查询模块索引缓冲池
RPT_DBP	32K	16 GB	报表模块数据缓冲池
RPT_IBP	32K	8 GB	报表模块索引缓冲池

以 DBA 权限，执行下面建立缓冲池的语句：

```
--16 GB
CREATE BUFFERPOOL "HR_DBP"  SIZE 2048000 PAGESIZE 8192 NOT EXTENDED STORAGE
--8 GB
CREATE BUFFERPOOL "HR_IBP"  SIZE 1024000 PAGESIZE 8192 NOT EXTENDED STORAGE
--16 GB
CREATE BUFFERPOOL "QRY_DBP"  SIZE 1024000 PAGESIZE 16384 NOT EXTENDED STORAGE
--8 GB
CREATE BUFFERPOOL "QRY_IBP"  SIZE 512000 PAGESIZE 16384 NOT EXTENDED STORAGE
--16 GB
CREATE BUFFERPOOL "RPT_DBP"  SIZE 512000 PAGESIZE 32768 NOT EXTENDED STORAGE
--8 GB
CREATE BUFFERPOOL "RPT_IBP"  SIZE 256000 PAGESIZE 32768 NOT EXTENDED STORAGE
```

再次使用“db2pd –db erpdb -bufferpools”命令，可用看到内存已经按照新的规划正确分配，另外，缓冲池的命中率也达到了 99.99%~100%。

```
    Database Partition 0 -- Database ERPDB -- Active -- Up 5 days 23:19:55 -- Date 2013-01-27
14:44:16

    Bufferpools:
    First Active Pool ID      1
    Max Bufferpool ID        7
    Max Bufferpool ID on Disk 7
    Num Bufferpools          11

    Address           Id   Name           PageSz     PA-NumPgs  BA-NumPgs  BlkSize
NumTbsp    PgsToRemov CurrentSz  PostAlter  SuspndTSCt Automatic
    0x000000000B2F93A0 1    IBMDEFAULTBP     8192       1000       0          0          5
0         1000       1000       0          True
    0x000000000B2F9840 2    HR_DBP          8192       2048000     0          0          1
0         2048000     2048000     0          False
    0x000000000B2F9CE0 3    HR_IBP          8192       1024000     0          0          1
0         1024000     1024000     0          False
    0x000000000B2FA180 4    QRY_DBP         16384      1024000     0          0          1
0         1024000     1024000     0          False
```

```
    0x000000000B2FA620 5    QRY_IBP         16384       512000      0          0          1
0         512000      512000      0          False
    0x000000000B2FAAC0 6    RPT_DBP         32768       512000      0          0          1
0         512000      512000      0          False
    0x000000000B2FAF60 7    RPT_IBP         32768       256000     0          0          1
0          256000      256000     0          False
    0x000000000B2F8120 4096 IBMSYSTEMBP4K      4096       16         0          0          0
0         16        16         0          False
    0x000000000B2F85C0 4097 IBMSYSTEMBP8K      8192       16         0          0          0
0         16        16         0          False
    0x000000000B2F8A60 4098 IBMSYSTEMBP16K     16384      16         0          0          0
0         16        16         0          False
    0x000000000B2F8F00 4099 IBMSYSTEMBP32K     32768      16         0          0          0
0         16        16         0          False

    Bufferpool Statistics for all bufferpools (when BUFFERPOOL monitor switch is ON):

    BPID DatLRds    DatPRds    HitRatio TmpDatLRds TmpDatPRds HitRatio IdxLRds    IdxPRds
HitRatio TmpIdxLRds TmpIdxPRds HitRatio
    1    142234263  13827622   90.28%   14220952   809871     94.31%   127553113  1271518
99.00%   1736       0          100.00%
    2    4590134        0        100%   0          0          00.00%   20         6
70.00%   0          0          00.00%
    3    16553   0   100%   0          0          00.00%   0          0          00.00%   0
0         00.00%
    4    2804135    600        99.99%   0          0          00.00%   0          0
00.00%   0          0          00.00%
    5    167212     70       99.99%   0          0          00.00%   0          0
00.00%   0          0          00.00%
    6    288793        30         99.99%   0          0          00.00%   3108461    40819
98.69%   0          0          00.00%
    7    112513        3         99.99%   0          0          00.00%   28300983   174
100.00%   0          0          00.00%
    4096 0          0          00.00%   0          0          00.00%   0          0
00.00%   0          0          00.00%
    4097 0          0          00.00%   0          0          00.00%   0          0
00.00%   0          0          00.00%
    4098 0          0          00.00%   0          0          00.00%   0          0
00.00%   0          0          00.00%
    4099 0          0          00.00%   0          0          00.00%   0          0
00.00%   0          0          00.00%
```

（2）排序堆配置

DB2 中的排序分为共享排序和私有排序，这主要是由 SHEAPTHRES 这个实例级参数控制。如果 SHEAPTHRES 被设置为 0 值，则使用共享排序，这时 SHEAPTHRES_SHR 将起作用，其值代表共享排序的最大内存限制；如果 sheapthres 被设置为非 0 值，表示 DB2 将使用私有排序，排序内存在代理中分配，SORTHEAP 参数此时则规定了每个私有排序最多使用的内存量。

首先使用快照命令，查看排序溢出情况。从下面可以看出，总计 8673578 次排序，但是发生了 208840 次溢出，明显偏多。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
已分配的专用排序堆总数            = 0
已分配的共享排序堆总数                    = 0
共享排序堆高水位标记                      = 920752
后阈值排序（共享内存）                    = 90
总计排序                                     = 8673578
总计排序时间（毫秒）                         = 1104503
排序溢出                                     = 208840
活动排序数                        = 0
...
```

那么怎么来调整呢？首先，先看一下原来的设置，SHEAPTHRES 设置为 0，表示启用共享排序，但是 SHEAPTHRES_SHR 为 AUTOMATIC(2048)，SORTHEAP 为 AUTOMATIC(1024)，明显偏小了，无法满足应用的需求。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get dbm cfg  | grep -i 'sort'
 Sort heap threshold (4KB)                  (SHEAPTHRES) = 0
[erpinst1@PrimaryNode-2 ~]$ db2 get db cfg  | grep -i 'sort'
 Sort heap thres for shared sorts (4KB) (SHEAPTHRES_SHR) = AUTOMATIC(2048)
 Sort list heap (4KB)                         (SORTHEAP) = AUTOMATIC(1024)
```

增加排序堆大小的命令如下：

```
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg using sheapthres_shr 1638400
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg using sortheap 819200
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

调整后的大小如下，运行一段时间重新执行快照命令，发现排序溢出次数大大减少了，从原来的 208840 次减少为 2312 次。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
已分配的专用排序堆总数            = 0
已分配的共享排序堆总数                    = 0
共享排序堆高水位标记                      = 920752
后阈值排序（共享内存）                    = 90
总计排序                                     = 717534
总计排序时间（毫秒）                         = 901304
排序溢出                                     = 2312
活动排序数                        = 0
...
```

（3）包缓存配置

包缓存保存了最近使用的 SQL 语句或者存储过程的查询计划，如果有相同的查询，可以直接从包缓存中获得查询计划，而不需要重新编译。所以，包缓存的大小设定非常重要。对于 OLAP 应用来说，重复的查询相对较少，而且查询语句相对较大，有少量的溢出应该不是问题，但是对于 OLTP 来说，就是很大的问题了。这可以通过快照命令获取包缓存溢出情况，如下所示，发现有 2103 次溢出。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
程序包高速缓存查询                              = 6705019
程序包高速缓存插入                              = 783087
程序包高速缓存溢出                            = 2103
程序包高速缓存高水位标记（以字节计）          = 407662554
应用程序节查找                                  = 13403852
应用程序节插入                                  = 944293
...
```

查看原先 PCKCACHESZ 设置，其值为 AUTOMATIC(1024)，明显偏小：

```
[erpinst1@PrimaryNode-2 ~]$ db2 get db cfg | grep -i 'package'
 Package cache size (4KB)                (PCKCACHESZ) = AUTOMATIC(1024)
```

增加包缓存大小，从 1024 增加为 102400：

```
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg using PCKCACHESZ 102400
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

运行一段时间后，重新执行快照命令，发现溢出次数大为减小，只有 103 次。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
程序包高速缓存查询                              = 5203029
程序包高速缓存插入                              = 1083087
程序包高速缓存溢出                            = 103
程序包高速缓存高水位标记（以字节计）          = 7362414
应用程序节查找                                  = 12603752
应用程序节插入                                  = 834201
...
```

（4）编目缓存配置

编目缓存包含最近用到的数据库编目信息，包括表的定义、统计信息和权限等。这可以通过快照命令获取包缓存溢出情况，如下所示，发现有 901 次溢出：

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
目录高速缓存查询                                = 2626178
目录高速缓存插入                                = 321
目录高速缓存溢出                                = 901
目录高速缓存高水位标记                          = 1554280
...
```

查看原先 CATALOGCACHE_SZ 设置，其值为 512，明显偏小，导致多次溢出：

```
[erpinst1@PrimaryNode-2 ~]$ db2 get db cfg | grep -i 'catalog'
Catalog cache size (4KB)             (CATALOGCACHE_SZ) = 512
```

增加编目缓存大小，从 512 增加为 102400：

```
[erpinst1@PrimaryNode-2 ~]$ db2 update db cfg using CATALOGCACHE_SZ 102400
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

运行一段时间后，重新执行快照命令，发现溢出次数减为 0 次。

```
[erpinst1@PrimaryNode-2 ~]$ db2 get snapshot for db on erpdb
...
目录高速缓存查询                          = 1526178
目录高速缓存插入                          = 212
目录高速缓存溢出                          = 0
目录高速缓存高水位标记                    = 1256193
...
```

3. 根据冷热程度灵活调整表

对性能优化来说，表的定义通常不是 DBA 关注的重点，他们最关注表的冷热程度，其次是表的大小。如表 2-9 所示，可以分为冷大表、热大表、冷小表和热小表。在性能调优中，最关注的是热大表和热小表。

表 2-9 缓冲池页面分配情况和命中率

表	大	小
冷	冷大表	冷小表
热	热大表	热小表

那么怎么来找到数据库中的热表呢？可以通过“db2pd -db erpdb -tcbstats”命令查看数据库运行一段时间内，表的大小和冷热情况，如下所示，CHARGE_HISTORY 有 51492320 个页面，显然是大表了，查询次数为 9269223，插入次数为 3139368，更新次数为 5844121，删除次数为 720323，所以也是热表了，总体来看，CHARGE_HISTORY 是一个供频繁查询、插入、更新和删除的热大表。

另外，也关注一下表 CHARGEDETERMINATION，是个小表，只有 5247 个页面，但是对这个表的查询很多，达到了 9701900 次，但是对这个表没有其他插入、更新和删除操作，也就是说这个表是一个只供查询的热小表。

```
    [erpinst1@PrimaryNode-2 ~]$ db2pd -db erpdb -tcbstats
    Database Partition 0 -- Database ERPDB -- Active -- Up 5 days 22:49:19 -- Date 2013-01-27
14:13:40

    TCB Table Information:
    Address            TbspaceID TableID PartID MasterTbs MasterTab TableName
    SchemaNm ObjClass DataSize   LfSize    LobSize    XMLSize
    ...
    0x00000003CE701828 4         13      n/a    4         13        CHARGEDETERMINATION
QRY    Perm     5267    0          0          0
    0x00000003AE4C0DA8 4         15      n/a    4         15        CHARGE_HISTORY
QRY    Perm     51492320     0          0          0
    ..
    TCB Table Stats:
    Address            TableName          Scans      UDI        PgReorgs   NoChgUpdts Reads
FscrUpdates Inserts    Updates    Deletes    OvFlReads  OvFlCrtes
    ...
    0x00000003CE701828 CHARGEDETERMINATION    9         27610676   0          0
```

```
9701900          0          0          0          0          0          0
    0x00000003AE4C0DA8 CHARGE_HISTORY            0          9703812    85737      0
9269223    445751     3139368    5844121    720323     0          0
    ...
```

（1）调整热大表 CHARGE_HISTORY

首先，看一下 CHARGE_HISTORY 的表定义，字段比较多，其中 CHARGEIDENTIFIER 为主键，如下所示。另外，和 DBA 沟通过，这个表中的字段 BILLINGPERIODENDDATE 表示每天记录执行操作时的月份。

```
    CREATE TABLE "QRY"."CHARGE_HISTORY"  (
              "CHARGEIDENTIFIER" BIGINT NOT NULL ,
              "CHARGEAMOUNTINVOICEABLE" DOUBLE NOT NULL ,
              "CHARGEAMOUNTOPPORTUNITY" DOUBLE NOT NULL ,
              ...
              "WORKSTATIONIDENTIFIER" CHAR(10) NOT NULL ,
              "ORIGINALEXTERNALINVOICENUMBER" CHAR(30) ,
              "BILLINGPERIODENDDATE" DATE )
              IN "QRY_DTS" INDEX IN "QRY_ITS"

    ALTER TABLE "QRY"."CHARGE_HISTORY"
        ADD PRIMARY KEY
            ("CHARGEIDENTIFIER");
```

那么怎么解决这个问题呢？CHARGE_HISTORY 改为分区表，例如以 BILLINGPERIODENDDATE 作为表分区的键值，按月进行分区，将很少用的数据放到慢速的内置硬盘（路径为“/hdd”），将有时会用到的数据放到磁盘阵列（路径为“/raid”），将频繁使用的数据放到固态盘（路径为“/ssd”）。

先看看在 DB2 V9.7 中是怎么实现的：

```
    CREATE BUFFERPOOL "cold_bp"  SIZE 128000 PAGESIZE 8192 NOT EXTENDED STORAGE;
    CREATE BUFFERPOOL "warm_bp"  SIZE 128000 PAGESIZE 8192 NOT EXTENDED STORAGE;
    CREATE BUFFERPOOL "hot_bp"  SIZE 128000 PAGESIZE 8192 NOT EXTENDED STORAGE;
    CREATE LARGE TABLESPACE "tbsp_2012_4Q" PAGESIZE 8192 MANAGED BY DATABASE
         USING (FILE '/hdd/20124Q01.DAT'2097152,
            FILE '/hdd/20124Q02.DAT'2097152,
            FILE '/hdd/20124Q03.DAT'2097152,
            FILE '/hdd/20124Q04.DAT'2097152
            )
         EXTENTSIZE 32
         PREFETCHSIZE 64
         BUFFERPOOL cold_bp
         OVERHEAD 7.500000
         TRANSFERRATE 0.060000
         NO FILE SYSTEM CACHING;
    CREATE LARGE TABLESPACE "tbsp_2013_1Q" PAGESIZE 8192 MANAGED BY DATABASE
         USING (FILE '/raid/20131Q01.DAT'2097152,
            FILE '/raid/20131Q02.DAT'2097152,
            FILE '/raid/20131Q03.DAT'2097152,
            FILE '/raid/20131Q04.DAT'2097152
            )
```

```
      EXTENTSIZE 32
      PREFETCHSIZE 64
      BUFFERPOOL warm_bp
      OVERHEAD 7.500000
      TRANSFERRATE 0.060000
      NO FILE SYSTEM CACHING;
CREATE LARGE TABLESPACE "tbsp_2013_2Q" PAGESIZE 8192 MANAGED BY DATABASE
      USING (FILE '/hdd/20132Q01.DAT'2097152,
          FILE '/hdd/20132Q02.DAT'2097152,
          FILE '/hdd/20132Q03.DAT'2097152,
          FILE '/hdd/20132Q04.DAT'2097152
          )
      EXTENTSIZE 32
      PREFETCHSIZE 64
      BUFFERPOOL hot_bp
      OVERHEAD 7.500000
      TRANSFERRATE 0.060000
```

在 DB2 V10 中，实现的方法将更为简单，如下所示：

```
-- DB2 V10 的办法
CREATE STOGROUP sg_hot ON '/hdd'
CREATE STOGROUP sg_warm ON '/raid'
CREATE STOGROUP sg_warm ON '/ssd'
CREATE TABLESPACE tbsp_2012_4Q USING STOGROUP sg_cold
CREATE TABLESPACE tbsp_2013_1Q USING STOGROUP sg_warm
CREATE TABLESPACE tbsp_2013_2Q USING STOGROUP sg_hot
```

最终，将 CHARGE_HISTORY 改造为分区表后的 DDL 定义如下所示，BILLINGPERIODENDDATE 为分区键，其中 2012 年第 4 季度的数据存放在 2012_tbsp_4Q 表空间上，2013 年第 1 季度的数据存放在 2013_tbsp_1Q 表空间上，2013 年第 2 季度的数据存放在 2013_tbsp_2Q 表空间上。

```
        CREATE TABLE "QRY"."CHARGE_HISTORY"  (
         "CHARGEIDENTIFIER" BIGINT NOT NULL ,
         "CHARGEAMOUNTINVOICEABLE" DOUBLE NOT NULL ,
         "CHARGEAMOUNTOPPORTUNITY" DOUBLE NOT NULL ,
         ...
         "WORKSTATIONIDENTIFIER" CHAR(10) NOT NULL ,
         "ORIGINALEXTERNALINVOICENUMBER" CHAR(30) ,
         "BILLINGPERIODENDDATE" DATE )
        INDEX IN "QRY_ITS" PARTITION BY RANGE("BILLINGPERIODENDDATE" NULLS FIRST)
        (PART "PART0" STARTING(MINVALUE) IN "2012_tbsp_4Q",
        PART "PART1" STARTING('2012-10-01') IN "2012_tbsp_4Q",
        PART "PART2" STARTING('2012-11-01') IN "2012_tbsp_4Q",
        PART "PART3" STARTING('2012-12-01') IN "2012_tbsp_4Q",
        PART "PART4" STARTING('2013-01-01') IN "2013_tbsp_1Q",
        PART "PART5" STARTING('2013-02-01') IN "2013_tbsp_1Q",
        PART "PART6" STARTING('2013-03-01') IN "2013_tbsp_1Q",
        PART "PART7" STARTING('2013-04-01') IN "2013_tbsp_2Q",
        PART "PART8" STARTING('2013-05-01') IN "2013_tbsp_2Q",
        PART "PART9" STARTING('2013-06-01') IN "2013_tbsp_2Q",
        PART "PART10" STARTING('2013-07-01') ENDING('2013-08-01') EXCLUSIVE IN
"2013_tbsp_2Q")
```

（2）调整热小表 CHARGEDETERMINATION

首先查看一下这个表定义，如下所示：

```
CREATE TABLE "QRY"."CHARGEDETERMINATION"  (
          "ACTIVERECORD" CHAR(1) NOT NULL ,
          "AIRCRAFTCLASS" CHAR(2) NOT NULL WITH DEFAULT '*' ,
          "ENGINETYPE" CHAR(1) NOT NULL WITH DEFAULT '*' ,
          "AIRCRAFTOPERATOR" CHAR(8) NOT NULL WITH DEFAULT '*' ,
          "AIRCRAFTOPERATORTYPE" CHAR(4) NOT NULL WITH DEFAULT '*' ,
          "AIRCRAFTTYPE" CHAR(15) NOT NULL WITH DEFAULT '*' ,
          "AIRCRAFTUSAGE" CHAR(2) NOT NULL ,
          "ANCILLARYFLIGHTTYPE" CHAR(2) NOT NULL WITH DEFAULT '*' ,
          "ANCILLARYROUTETYPE" CHAR(1) NOT NULL ,
          "LOWERBOUNDARY1" INTEGER ,
          "UPPERBOUNDARY1" INTEGER ,
          "LOWERBOUNDARY2" INTEGER ,
          "UPPERBOUNDARY2" INTEGER ,
          "CALCULATIONREFERENCE" INTEGER NOT NULL ,
          "CHANGEREASON" CHAR(4) ,
          "NOTES" VARCHAR(1000) ,
          "DATAINTEGRITYINDICATOR" CHAR(1) NOT NULL ,
          "DATETIMESTAMP" TIMESTAMP NOT NULL WITH DEFAULT CURRENT TIMESTAMP ,
          "ENDDATE" TIMESTAMP NOT NULL ,
          "STARTDATE" TIMESTAMP NOT NULL ,
          "FILEDFLIGHTRULES" CHAR(2) NOT NULL ,
          "FINALFLIGHTRULES" CHAR(2) NOT NULL ,
          "FLIGHTTYPEICAO" CHAR(3) NOT NULL WITH DEFAULT '*' ,
          "NOISECATEGORY" CHAR(8) NOT NULL WITH DEFAULT '*' ,
          "NUMBERENGINES" CHAR(1) NOT NULL WITH DEFAULT '*' ,
          "PROGRAMIDENTIFIER" CHAR(10) NOT NULL ,
          "ROUTETYPE" CHAR(2) NOT NULL WITH DEFAULT '*' ,
          "SERVICECODE" CHAR(10) NOT NULL ,
          "SERVICELEVELCODE" CHAR(2) NOT NULL WITH DEFAULT '*' ,
          "SERVICELOCATION" CHAR(6) NOT NULL WITH DEFAULT '*' ,
          "SERVICEPROVIDER" CHAR(8) NOT NULL ,
          "TICKETABLE" CHAR(1) NOT NULL WITH DEFAULT 'N' ,
          "TICKETCLIPS" INTEGER WITH DEFAULT 0 ,
          "UNIQUEIDENTIFIER" INTEGER NOT NULL ,
          "USERIDENTIFIER" CHAR(10) NOT NULL ,
          "WORKSTATIONIDENTIFIER" CHAR(10) NOT NULL )
         IN "QRY_DTS" INDEX IN "QRY_ITS"

ALTER TABLE "QRY"."CHARGEDETERMINATION"
    ADD PRIMARY KEY
        ("SERVICECODE",
         "SERVICEPROVIDER",
         "STARTDATE")
```

接下来，从管理视图“sysibmadm.snapdyn_sql”中查询一下排名前十位执行时间最多的查询语句，可以看到“select ... from QRY.CHARGEDETERMINATION q1 where（q1."SERVICECODE" = ?）and（q1."SERVICEPROVIDER" = ?） and （q1."STARTDATE" <= ?）”这条语句是执行时

间最长的。如下所示：

```
    select  NUM_EXECUTIONS, TOTAL_EXEC_TIME, STMT_TEXT from sysibmadm.snapdyn_sql order
by totaltime desc fetch first 10 rows only
    NUM_EXECUTIONS       TOTAL_EXEC_TIME      STMT_TEXT
    -------------------- --------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
--------            6866230              201235 select  q1."ACTIVERECORD",
q1."AIRCRAFTCLASS",  q1."ENGINETYPE",  q1."AIRCRAFTOPERATOR",
q1."AIRCRAFTOPERATORTYPE",  q1."AIRCRAFTTYPE",  q1."AIRCRAFTUSAGE",
q1."ANCILLARYFLIGHTTYPE",  q1."ANCILLARYROUTETYPE",  q1."LOWERBOUNDARY1",
q1."UPPERBOUNDARY1",  q1."LOWERBOUNDARY2",  q1."UPPERBOUNDARY2",
q1."CALCULATIONREFERENCE",  q1."CHANGEREASON",  q1."NOTES",
q1."DATAINTEGRITYINDICATOR",  q1."DATETIMESTAMP",  q1."ENDDATE",  q1."STARTDATE",
q1."FILEDFLIGHTRULES",  q1."FINALFLIGHTRULES",  q1."FLIGHTTYPEICAO",
q1."NOISECATEGORY",  q1."NUMBERENGINES",  q1."PROGRAMIDENTIFIER",  q1."ROUTETYPE",
q1."SERVICECODE",  q1."SERVICELEVELCODE",  q1."SERVICELOCATION",  q1."SERVICEPROVIDER",
q1."TICKETABLE",  q1."TICKETCLIPS",  q1."UNIQUEIDENTIFIER",  q1."USERIDENTIFIER",
q1."WORKSTATIONIDENTIFIER" from QRY.CHARGEDETERMINATION q1 where  ( q1."SERVICECODE" = ?)
and  ( q1."SERVICEPROVIDER" = ?)  and   ( q1."STARTDATE" <= ?)
    ...
```

那么怎么解决这个问题呢？在数据库中对这种静态热小表执行千万次查询，对性能有影响，有时候会成为瓶颈。下面的方法是重构应用来解决，即在应用中缓存这个表，从而避免对数据库执行大量查询操作。

如下所示的示意代码，使用 Java 类 ChargeDeterminationCache 封装对表 ChargeDetermination 的操作。其中使用 hashmap 保存表 CHARGEDETERMINATION 中的数据，其中方法 initializeData 用来初始化表中数据到 hashmap 中，方法 getChargeDeterminationList 相当于上述的 SQL 查询。

```
    public class ChargeDeterminationCache {

        private static ChargeDeterminationCache instance;

    //使用 hashmap 保存表 CHARGEDETERMINATION 中的数据
        private HashMap<String, List<Chargedetermination>> chargeDeterminationMap;

        private ChargeDeterminationCache() {
            initializeData ();
        }

        public HashMap<String, List<Chargedetermination>> getCache() {
            return chargeDeterminationMap;
        }
    /***
    调用这个方法，相当于执行上述查询 select ... from QRY.CHARGEDETERMINATION q1 where
    ( q1."SERVICECODE" = ?)  and  ( q1."SERVICEPROVIDER" = ?)  and   ( q1."STARTDATE"
    <= ?)
    ***/
        public static List<Chargedetermination> getChargeDeterminationList(
                String serviceCode, String serviceProvider, Timestamp when)
```

```
                throws FinderException {
            HashMap<String, List<Chargedetermination>> chargeDeterminationMap =
getInstance()
                    .getCache();
            String key = serviceProvider.trim() + serviceCode.trim();
            if (chargeDeterminationMap.containsKey(key)) {
                List<Chargedetermination> result = new
ArrayList<Chargedetermination>();
                List<Chargedetermination> list = chargeDeterminationMap.get(key);
                for (Chargedetermination dto : list) {
                    if (dto.getStartDate().compareTo(when) <= 0
                            && dto.getEndDate().compareTo(when) >= 0) {
                        result.add(dto);
                    }
                }
                if (result.size() > 0) {
                    return result;
                }
            }
            throw new FinderException();
        }
    //将 CHARGEDETERMINATION 表中数据初始化到 hashmap 中
        private void initializeData() {
            chargeDeterminationMap = new HashMap<String, List<Chargedetermination>>();
            Connection conn = null;
            Statement stmt = null;
            ResultSet rs = null;
            try {
                conn = getConnection();
                stmt = conn.createStatement();
                rs = stmt
                        .executeQuery("select * from CHARGEDETERMINATION");
                while (rs.next()) {
                    Chargedetermination chargedetermination = new Chargedetermination(
                            rs.getString("activerecord"),
                            rs.getString("aircraftclass"),
                            rs.getString("enginetype"),
                            rs.getString("aircraftoperator"),
                            rs.getString("aircraftoperatortype"),
                            rs.getString("aircrafttype"),
                            rs.getString("aircraftusage"),
                            rs.getString("ancillaryflighttype"),
                            rs.getString("ancillaryroutetype"),
                            rs.getInt("lowerboundary1"),
                            rs.getInt("upperboundary1"),
                            rs.getInt("lowerboundary2"),
                            rs.getInt("upperboundary2"),
                            rs.getInt("calculationreference"),
                            rs.getString("changereason"),
                            rs.getString("notes"),
                            rs.getString("dataintegrityindicator"),
                            rs.getTimestamp("datetimestamp"),
                            rs.getTimestamp("enddate"),
```

```
                                rs.getTimestamp("startdate"),
                                rs.getString("filedflightrules"),
                                rs.getString("finalflightrules"),
                                rs.getString("flighttypeicao"),
                                rs.getString("noisecategory"),
                                rs.getString("numberengines"),
                                rs.getString("programidentifier"),
                                rs.getString("routetype"),
                                rs.getString("servicecode").trim(),
                                rs.getString("servicelevelcode"),
                                rs.getString("servicelocation"),
                                rs.getString("serviceprovider").trim(),
                                rs.getString("ticketable"),
                                rs.getInt("ticketclips"),
                                rs.getInt("uniqueidentifier"),
                                rs.getString("useridentifier"),
                                rs.getString("workstationidentifier"));
                      String key = chargedetermination.getServiceProvider()
                              + chargedetermination.getServiceCode();
                      if (chargeDeterminationMap.containsKey(key)) {
                          List<Chargedetermination> chargedeterminations =
chargeDeterminationMap
                                   .get(key);
                          chargedeterminations.add(chargedetermination);
                      } else {
                          List<Chargedetermination> chargedeterminations = new
ArrayList<Chargedetermination>();
                          chargedeterminations.add(chargedetermination);
                          chargeDeterminationMap.put(key, chargedeterminations);
                      }
                  }

              } catch (SQLException e) {
                  e.printStackTrace();
              }finally{
                  try {
                      if(rs!=null)rs.close();
                      if(stmt!=null)stmt.close();
                      if(conn!=null)conn.close();
                  } catch (SQLException e) {
                      e.printStackTrace();
                  }

              }
          }

     }
```

2.5.2 人力资源模块优化

在本书 1.4.6 小节，我们谈到“懒惰系统”问题。业务繁忙时，通过对人力资源模块的监控，发现系统没有明显的 CPU、磁盘、内存或网络瓶颈，但是事务的响应时间比平时要长很多，

平时不到 1s 就能处理了，但是忙时可以达到 3s 以上，这个正是“懒惰系统”的症状。

这种现象通常是和锁有关，而且通常是死锁，从 db2diag.log 里面发现了 900 多次死锁事件。

小技巧：使用 DB2 自带的死锁检测器

为了方便 DBA 或者开发人员更好地检测和处理死锁，DB2 提供了一个默认的死锁检测器 db2detaildeadlock，只要执行“db2evmon -db erpdb -evm db2detaildeadlock”命令，随后根据输出就可以方便地分析和定位死锁了。

下面 db2evmon 命令生成的片段，发现和 VM_CODE 这个表有关，其中死锁是由于“UPDATE HR.VM_CODE SET MARKET_CODE = ?, BATCH_NO = ?, SERIAL_NO = ? WHERE CODE_ID = ?”这个 update 语句导致的。

```
db2evmon -db casms -evm db2detaildeadlock
...
4067) Deadlock Event ...
  Deadlock ID:   1323
  Number of applications deadlocked: 2
  Deadlock detection time: 2012-11-10 15:37:54.247477
  Rolled back Appl participant no: 2
  Rolled back Appl Id: 10.5.1.19.8968.121109204404
  Rolled back Appl seq number: : 1b74

4068) Deadlocked Connection ...
  Deadlock ID:   1323
  Participant no.: 2
  Participant no. holding the lock: 1
  Appl Id: 10.5.1.19.8968.121109204404
  Appl Seq number: 07028
  Appl Id of connection holding the lock: 10.5.1.19.61192.121109232601
  Seq. no. of connection holding the lock: 00014
  Lock wait start time: 2012-11-10 15:37:45.184779
  Lock Name       : 0x04000F001100EB588400000052
  Lock Attributes : 0x00000000
  Release Flags   : 0x40000001
  Lock Count      : 1
  Hold Count      : 0
  Current Mode    : NS  - Share (and Next Key Share)
  Deadlock detection time: 2012-11-10 15:37:54.247572
  Table of lock waited on      : VM_CODE
  Schema of lock waited on     : HR
  Data partition id for table  : 0
  Tablespace of lock waited on : HR_DTS
  Type of lock: Row
  Mode of lock: NS  - Share (and Next Key Share)
  Mode application requested on lock: X  - Exclusive
  Node lock occured on: 0
  Lock object name: 568427479057
  Application Handle: 508
```

```
    Deadlocked Statement:
      Type     : Dynamic
      Operation: Execute
      Section  : 3
      Creator  : NULLID
      Package  : SYSSN300
      Cursor   : SQL_CURSN300C3
      Cursor was blocking: FALSE
      Text     : UPDATE HR.VM_CODE  SET MARKET_CODE = ?, BATCH_NO = ?, SERIAL_NO = ?  WHERE
CODE_ID = ?
    List of Locks:
       ...

        Lock Name                   : 0x04000F001100EB588400000052
        Lock Attributes             : 0x00000000
        Release Flags               : 0x40000001
        Lock Count                  : 1
        Hold Count                  : 0
        Lock Object Name            : 568427479057
        Object Type                 : Row
        Tablespace Name             : HR_DTS
        Table Schema                : HR
        Table Name                  : VM_CODE
        Data partition id           : 0
        Mode                        : X   - Exclusive
        Status                      : Converting
        Current Mode                : NS  - Share (and Next Key Share)

        Lock Name                   : 0x04000F00000000000000000054
        Lock Attributes             : 0x00000000
        Release Flags               : 0x40000001
        Lock Count                  : 2
        Hold Count                  : 0
        Lock Object Name            : 15
        Object Type                 : Table
        Tablespace Name             : HR_DTS
        Table Schema                : HR
        Table Name                  : VM_CODE
        Data partition id           : 0
        Mode                        : IX  - Intent Exclusive
        ...
    Locks Held:    20
    Locks in List: 20
    Locks Displayed: 20

  4069) Deadlocked Connection ...
    Deadlock ID:   1323
    Participant no.: 1
    Participant no. holding the lock: 2
    Appl Id: 10.5.1.19.61192.121109232601
    Appl Seq number: 05342
    Appl Id of connection holding the lock: 10.5.1.19.8968.121109204404
```

```
      Seq. no. of connection holding the lock: 00001
      Lock wait start time: 2012-11-10 15:37:45.127741
      Lock Name      : 0x04000F001100EB588400000052
      Lock Attributes : 0x00000000
      Release Flags   : 0x40000060
      Lock Count     : 2
      Hold Count     : 0
      Current Mode    : NS  - Share (and Next Key Share)
      Deadlock detection time: 2012-11-10 15:37:54.248866
      Table of lock waited on      : VM_CODE
      Schema of lock waited on     : HR
      Data partition id for table  : 0
      Tablespace of lock waited on : HR_DTS
      Type of lock: Row
      Mode of lock: NS  - Share (and Next Key Share)
      Mode application requested on lock: X   - Exclusive
      Node lock occured on: 0
      Lock object name: 568427479057
      Application Handle: 314
      Deadlocked Statement:
        Type     : Dynamic
        Operation: Execute
        Section  : 4
        Creator  : NULLID
        Package  : SYSSN300
        Cursor   : SQL_CURSN300C4
        Cursor was blocking: FALSE
        Text     : UPDATE HR.VM_CODE  SET MARKET_CODE = ?, BATCH_NO = ?, SERIAL_NO = ?  WHERE
CODE_ID = ?
      List of Locks:
          ...
          Lock Name                 : 0x04000F000500EB588400000052
          Lock Attributes            : 0x00000000
          Release Flags              : 0x00000020
          Lock Count                : 1
          Hold Count                : 0
          Lock Object Name           : 568427479045
          Object Type               : Row
          Tablespace Name            : HR_DTS
          Table Schema              : HR
          Table Name                : VM_CODE
          Data partition id          : 0
          Mode                     : NS  - Share (and Next Key Share)
          ...

      Locks Held:    20
      Locks in List: 20
      Locks Displayed: 20
```

当第一次发现这个问题时，第一个建议是让 DBA 去检查应用服务器 Websphere 应用服务器中的 JDBC 数据源的隔离级别，从读稳定性（RS）调整为游标稳定性（CS），本来以为调整后可以解决这个问题，但是最终结果是没有太大作用。

注意：Websphere 6.0 应用服务器连接 DB2 的默认隔离级别

Websphere 应用服务器 6.0 或者以下版本，连接 DB2 数据库时，默认的隔离级别是读稳定性。在很多应用中，读稳定性过于严格了，为了提高并发性，建议将其调整为游标稳定性。

在解决锁问题的时候，如果能有开发人员和 DBA 的支持和配合，将会有助于更快地解决问题。在他们的配合下，使用下面的步骤最终解决了这个问题。

（1）查看 VM_CODE 表的 DDL 定义

可以看出这个表的字段很少，主键也很简单，只有一个字段 CODE_ID 作为主键。

```
CREATE TABLE "VMARKET "."VM_CODE"  (
          "CODE_ID" DECIMAL(15,0) NOT NULL ,
          "MARKET_CODE" VARCHAR(32) NOT NULL ,
          "BATCH_NO" DECIMAL(7,0) NOT NULL ,
          "SERIAL_NO" DECIMAL(7,0) NOT NULL)
         IN "HR_DTS" ;
-- DDL Statements for Primary Key on Table "VMARKET "."VM_CODE"

ALTER TABLE "VMARKET "."VM_CODE"
    ADD CONSTRAINT "P_KEY_1" PRIMARY KEY
        ("CODE_ID");
```

（2）了解应用逻辑

在人力资源管理模块中，每一个事务都会访问这个表 VM_CODE，首先根据 CODE_ID 查询出当前字段的值，包括 SERIAL_NO、MARKET_CODE 和 BATCH_NO 字段，其中最重要的是 SERIAL_NO 字段；用完后，使用 update 语句修改 SERIAL_NO 的值为新值。这就是典型的“先查再改”引起的锁问题，如下所示：

```
--开始事务
SELECT SERIAL_NO ,MARKET_CODE, BATCH_NO  FROM VM_CODE WHERE CODE_ID =?
...
--如果遇到问题，事务回滚
UPDATE VM_CODE  SET MARKET_CODE = ?, BATCH_NO = ?, SERIAL_NO = ?  WHERE CODE_ID = ?
--提交事务
```

（3）分析表中数据

但是奇怪的是 VM_CODE 这个表中竟然只有一条记录！！！如表 2-10 所示。这样一来，当业务繁忙时，所有的事务都在争抢这一条记录！因此业务繁忙时事务的响应时间就急剧增加了，而且事务吞吐量也无法提高！

表 2-10　VM_CODE 表中仅有一条记录

CODE_ID	MARKET_CODE	BATCH_NO	SERIAL_NO
's0001'	's0010001'	1	100002

（4）问题解决

如下所示，在表 VM_CODE 中新加入一个字段 HR_ID，这个字段用来标示人力资源部门

员工的 ID，这样的话，每个员工都有一个 HR_ID，使用自己的 SERIAL_NO。这样，当业务繁忙时，每个事务不会争抢同一条记录。

```
    CREATE TABLE "VMARKET "."VM_CODE"  (
              "CODE_ID" DECIMAL(15,0) NOT NULL ,
              "MARKET_CODE" VARCHAR(32) NOT NULL ,
              "BATCH_NO" DECIMAL(7,0) NOT NULL ,
              "SERIAL_NO" DECIMAL(7,0) NOT NULL,
              "HR_ID" VARCHAR(32) NOT NULL)
             IN "HR_DTS" ;
```

如表 2-11 所示，增加一个字段后，表 VM_CODE 中每一行记录将只为一个 HR_ID 所有，在进行事务处理时，每个 HR_ID 只更新自己所属的 SERIAL_NO 字段值，这样就避免了不同的事务争抢同一条记录。

表 2-11　重新设计后的 VM_CODE 表中数据

CODE_ID	MARKET_CODE	BATCH_NO	SERIAL_NO	HR_ID
's0001'	's0010001'	1	100002	'John'
's0002'	's0010001'	1	9001	'William'
's0003'	's0010001'	1	8900	'Mike'
...	...	...	...	...

（5）最终效果

如表 2-12 所示，调整后，事务响应时间从原来的 3s 减少为 1s 以下，而且事务吞吐量从 300 提高为 1500，性能得到了极大提升。

表 2-12　调整前后对比（忙时）

项目	事务响应时间（单位：秒）	事务吞吐量（单位：每秒事务量）
调整前	3s 以上	300
调整后	1s 以下	1500

2.5.3　查询分析模块优化

在 2.4.1 节我们讲过，这个模块的优化需求是一些动态查询语句要在 2 分钟以内出结果，那么怎么更有效地找到这些语句呢？这可以从最耗时或者执行最频繁的 SQL 语句入手。

一个数据库运行的查询语句非常多，重点是分析执行时间最长的前 N 条语句，例如前十位的查询语句。这可以从管理视图“sysibmadm.snapdyn_sql”中获取，其中最耗时间的查询语句如下所示，总的执行时间花费了 7386134 秒，执行次数达到了 4072390 次！

```
    select  TOTAL_EXEC_TIME, NUM_EXECUTIONS, STMT_TEXT from sysibmadm.snapdyn_sql order
by TOTAL_EXEC_TIME desc fetch first 10 rows only
    NUM_EXECUTIONS      TOTAL_EXEC_TIME      STMT_TEXT
    -------------------- --------------------
-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
--------           4072390              7386134
```

```
    SELECT FLIGHTDETAILIDENTIFIER, FLIGHTIDENTIFIER, flightDepartureTime FROM
QRY.FLIGHTDETAIL WHERE flightNumber = ?  AND flightDepartureTime > ? AND flightDepartureTime
<= ? ORDER BY flightDepartureTime DESC  FOR READ ONLY
    ...
```

那么怎么来优化这条查询语句呢？通常的办法是使用 visual Explain 或者 db2exfmt 来分析这些 SQL 语句；随后使用 Design Advisor 工具给出优化建议。关于这方面的细节，读者可以参考本书第 8 章的相关内容。

下面我们谈一下具体的步骤：

（1）使用下面的命令生成访问计划（关于如何生成访问计划，请阅读第 9 章）。

```
    [erpinst1@PrimaryNode-2 ~]$  db2 "explain plan with snapshot for SELECT
FLIGHTDETAILIDENTIFIER, FLIGHTIDENTIFIER, flightDepartureTime FROM QRY.FLIGHTDETAIL WHERE
flightNumber = ?  AND flightDepartureTime > ? AND flightDepartureTime <= ? ORDER BY
flightDepartureTime DESC  FOR READ ONLY"
    DB20000I  SQL 命令成功完成。
```

（2）使用 db2exfmt 格式化输出，并将输出重定向到文件 queryplan.txt 中。

```
    [erpinst1@PrimaryNode-2 ~]$ db2exfmt -d erpdb -1 -o queryplan.txt
    DB2 Universal Database Version 9.7, 5622-044 (c) Copyright IBM Corp. 1991, 2009
    Licensed Material - Program Property of IBM
    IBM DATABASE 2 Explain Table Format Tool

    Connecting to the Database.
    Connect to Database Successful.
    Output is in queryplan.txt.
    Executing Connect Reset -- Connect Reset was Successful.
```

（3）查看 queryplan.txt 文件中的访问计划。如下所示，可以看到这个查询需要排序和表扫描，执行代价比较高，达到了 597.793。

```
    [erpinst1@PrimaryNode-2 ~]$ more queryplan.txt
    ...
    Access Plan:
    -----------
        Total Cost:       598.793
        Query Degree:         1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
           0.00923729
             TBSCAN
             (   2)
             598.793
             79.1608
               |
           0.00923729
```

```
            SORT
            (   3)
            598.793
            79.1608
              |
          0.00923729
            FETCH
            (   4)
            598.793
            79.1608
         /----+-----\
     129.598     2.71776e+006
     IXSCAN     TABLE: QRY
     (   5)   FLIGHTDETAIL
     15.1815         Q1
        2
       |
  2.71776e+006
 INDEX: QRY
   X5ACPR37_32
      Q1
```

（4）要想消除排序和表扫描，建立合适的索引可以解决这个问题。考虑到查询的谓词，建立下面的复合索引，包括字段 FLIGHTNUMBER 和 FLIGHTDEPARTURETIME。

```
    CREATE INDEX "QRY     "."X6ACPR37_32" ON "QRY     "."FLIGHTDETAIL"
             ("FLIGHTNUMBER" ASC,
              "FLIGHTDEPARTURETIME" ASC)
             MINPCTUSED 10
    ALLOW REVERSE SCANS;
```

创建完毕后，运行 runstats 命令，重新收集统计信息，这样优化器在生成访问计划的时候就可以用上索引了。

```
    [erpinst1@PrimaryNode-2 ~]$ db2  runstats on table QRY.FLIGHTDETAIL with distribution
on all columns and indexes all
    DB20000I  RUNSTATS 命令成功完成。
```

（5）对比访问计划。

在真正执行之前，下面是新生成的访问计划，可以看到新建立的索引已经被优化器用上了，索引扫描代替了表扫描和排序，执行成本也相应减少到了 15.2025！

```
    Access Plan:
    -----------
         Total Cost:        15.2025
         Query Degree:          1

                  Rows
                 RETURN
                 (   1)
                  Cost
                   I/O
```

```
             |
          0.00865317
            FETCH
            (   2)
            15.2025
            2.00878
        /-----+-----\
  0.00878028     2.52048e+006
    IXSCAN      TABLE: QRY
    (   3)    FLIGHTDETAIL
    15.1359          Q1
       2
      |
 2.52048e+006
INDEX: QRY
  X6ACPR37_32
      Q1
```

再对比一下优化前后的查询时间对比，如表 2-13 所示，从原来的 1 分钟减少到了 10 秒，大幅提高了性能。

表 2-13　优化前后对比

查询语句	记录数	字段数	执行时间
优化前	27177610	62	1 分钟
优化后	27177610	62	10 秒

最后，我们用类似的办法优化了其他查询语句，最终使得所有的查询语句都在 2 分钟以内出结果，这满足了业务对查询时间的需要。

小技巧：使用 Design Advisor 工具优化 SQL 语句

其实，除了使用 db2exfmt 分析查询计划，随后手工创建索引之外，还可以通过 Design Advisor 工具来实现。这个工具提供的建议可以和专家建议相媲美，如果存在有大量需要优化的 SQL 语句，使用工具可以显著提高效率。关于 Design Advisor 的使用，读者可以参考第 9 章相关内容。

2.5.4　报表应用优化

在报表模块里面，报表数据库有自己单独的一套表结构，其中所需要的数据都是从人力资源数据库中的相关表中抽取过来的，抽取方式采取日增量抽取的方式。抽取和加载所允许的时间窗口为每天晚上 12 点到第二天凌晨 8 点，总共为 9 个小时。在上线初期，完成每天的所有数据抽取和加载所需要的时间只有 5 个小时，9 个小时的时间窗口完全可以满足需要，过了几个月后，每天需要 10 个小时才能完成抽取和加载，现有的 9 个小时的时间窗口已经难以满足需要了，需要进行性能优化。

先来分析一下数据抽取和加载的存储过程脚本，例如下面是一个增量抽取的脚本，其中重要的语句是“INSERT INTO ... SELECT”，用来抽取 HR.CHARGE 中数据，并加载到报表数据

库中的 RPT.CHARGE_REPORT 表中。真正影响性能的关键点就在这里，因为使用“INSERT INTO ... SELECT”语句，表越大，那么需要的插入时间就越多，而且会写大量的日志，对日志设备造成读写压力。

```
    CREATE PROCEDURE RPT.SP_DM_QUANTITY (in MONTH_OFFSET INTEGER)

    P1: BEGIN

    DECLARE sql_code INTEGER;
    DECLARE sqlstring varchar(1000);  --sql 语句
    DECLARE sql_errm varchar(1000);
    DECLARE begin_date date;
    DECLARE end_date date;
    DECLARE stepname varchar(1000);
    declare sqlcode integer default 0;
    DECLARE sqlstate char(5);

    --设置增量抽取结束时间
    set end_date=(select  date(rtrim(char(year(current date ))) || '-' ||
rtrim(char(month(current timestamp ))) || '-1')-1 day from sysibm.sysdummy1);

    ...
    --插入数据
    INSERT INTO RPT.CHARGE_REPORT
    SELECT CHARGEIDENTIFIER,CASE WHEN SOURCEUOM1='DISTANCE'
        THEN  QUANTITYUOM1
    ELSE CASE WHEN SOURCEUOM2='DISTANCE'
          THEN QUANTITYUOM2
       else 0  END
    END  as DISTANCE FROM HR.CHARGE;
    commit;
    ...
    END $
```

解决这个问题的关键点是找到对“INSERT INTO…SELECT”语句的替代方案，要求能够加载大量数据，加载时间比较稳定，而且对日志设备没有太大性能压力，很多读者可能会想到使用 LOAD 方式来做，但是使用常规 LOAD 方式的时候首先需要让源表的数据落地成文件，随后再 LOAD 到目标表中，需要占用额外的空间。

比较：LOAD 和 INSERT 数据导入速度

LOAD 导入与 INSERT 不同，它并不是通过 SQL 语句的 INSERT 进行导入，而是把导入数据以数据页的形式直接存入数据库中。这就意味着摆脱了 SQL 语句的束缚，以一种物理方式把数据放入数据库中，并且在导入过程中不记录日志，特别适合大数据量导入。另外，LOAD 过程中会使用并行性，有多个进程或者线程进行处理，理论上讲速度会明显高于 INSERT。

我们在这台 x3850 机器上做过试验，对于一个有 7 千万条记录，大小为 50GB 的表进行 INSERT 大约用了 6 个小时，而同样的数据使用 LOAD 导入，只用了 1 个小时左右就完成了。

下面介绍一种 LOAD 的高级用法，即 LOAD…FROM CURSOR 的方式，它本质上也是 LOAD，但是中间不需要让源表的数据落地成文件，所以在程序中使用更为方便。这可以通过特殊的存储过程 ADMIN_CMD 来调用，从而把 LOAD…FROM CURSOR 嵌入到存储过程代码中。如下所示，使用 LOAD…FROM CURSOR 代替了“INSERT INTO…SELECT”语句。

```
    CREATE PROCEDURE RPT.SP_DM_QUANTITY (in MONTH_OFFSET INTEGER)
    P1: BEGIN

    DECLARE sql_code INTEGER;
    DECLARE sqlstring varchar(1000);  --sql 语句
    DECLARE sql_errm varchar(1000);
    DECLARE begin_date date;
    DECLARE end_date date;
    DECLARE stepname varchar(1000);
    declare sqlcode integer default 0;
    DECLARE sqlstate char(5);

    --设置增量抽取结束时间
    set end_date=(select  date(rtrim(char(year(current date ))) || '-' ||
rtrim(char(month(current timestamp ))) || '-1')-1 day from sysibm.sysdummy1);

    ...
    --插入数据
    call sysproc.admin_cmd('load from(SELECT INVOICEDETAILIDENTIFIER,0,
    SUM(CASE WHEN SOURCEUOM1=''DISTANCE''
        THEN  QUANTITYUOM1
    ELSE CASE WHEN SOURCEUOM2=''DISTANCE''
          THEN QUANTITYUOM2
       else 0  END
    END) FROM HR.CHARGE
    GROUP BY INVOICEDETAILIDENTIFIER,0) of cursor insert into RPT.CHARGE_REPORT
nonrecoverable ');
    commit;
    ...
    END $
```

如表 2-14 所示，对同样一个具有 43 个字段、850 多万条记录的表，改变加载方式后，性能提高了 150%。在具体实施中，也用到了 LOAD…FROM CURSOR 同时并行抽取加载多个表，这样一来，就可以最大限度地利用上了硬件资源，所需要的时间从原来的每天 10 个小时，缩短为 4 个小时就可以完成了。

表 2-14　两种抽取和加载时间对比

测试项目	记录数	字段数	加载时间
INSERT INTO…SELECT	8638200	43	25 分钟
LOAD…FROM CURSOR	8638200	43	8 分钟

2.5.5　性能优化总结

最终，我们完成了这个 DB2 经营分析库的优化工作，所采用的方法如表 2-15 所示。性能

优化是有方法的，就是要找到性能瓶颈，但同时又被称为艺术，就是其灵活性，要根据不同的硬件和应用情况对症下药，灵活处理。

表 2-15　优化方法总结

数据库和应用	优化方法
全局调整和优化	不能一概而论，不同的系统有不同的全局调整办法，但重点和难点在于做好存储规划和内存参数调整，同时能够根据冷热程度灵活处理热表和大表
人力资源管理	针对“懒惰系统”，重点是解决锁争抢问题。这种问题通常是由于应用的原因导致的，例如应用中的多个事务先查询再修改某一张热表，这样容易发生多个事务同时争抢同一条记录。优化方法是重新对表进行了设计，也修改了事务执行的逻辑代码
查询分析	抓住重点，集中精力优化最耗时和执行最频繁的前 N 条 SQL 语句，这样可以以最少的投入换来最大的性能提升
报表	报表中的数据通常来自于业务系统，使用 LOAD…FROM CURSOR 代替了“INSERT INTO…SELECT”语句，这样既可以抽取加载大量数据，也不用写日志，不会对日志设备读写造成很大压力，从而提升性能

2.6　小结

本章以某汽车制造商 ERP 应用的 Oracle RAC 数据库系统为例，提出了扩展 DB2 经营分析库来对 Oracle RAC 进行“瘦身”的方案，随后给出实施步骤，并在此基础上提出了 DB2 HADR 方案来解决灾备问题。

上线运行一段时间出现性能问题后，又重点讲解了如何对其进行性能优化：即通过业务分析来确认性能问题，通过系统分析来确定性能原因，随后制定优化计划并成功实施。希望大家学习完这个案例后可以举一反三，真正指导自己的优化实践。

第3章 高质量物理设计

老赵是一家电信公司的 DBA，曾经维护过 Oracle、Sybase、Mysql 等多种数据库系统，今年开始建设基于 DB2 的客户关系管理系统（CRM），从而正式接触 DB2 了。前几天，他给我打来电话，“我的 DB2 数据库系统比蜗牛爬得都慢，帮帮我吧！”于是，我对老赵所在电信公司的 DB2 数据库进行了健康检查，从中发现了一系列问题。我和老赵进行了深入的技术交流，发现这些问题的最终根源在于系统建设初期不良的物理设计。

笔者前面批判过国内很多企业软硬件选型中的一个弊病，那就是盲目地选择高档硬件。但这个电信公司是另外一个极端，也太抠门了，所选择的 8 内核的 x3650 服务器已经落后多年，无法满足 CRM 系统对响应时间和事务吞吐量的要求。

除了硬件的抠门外，该电信公司在物理设计中也存在很多不合理的地方，例如没有一套针对热数据和大数据的存储和访问策略，没有选择合理的 RAID 级别，表空间和缓冲池的规划也不符合最佳实践。

用一句话概括，这是一个匆忙上马的项目，没有进行正规的硬件选型，没有进行合理的物理设计。

最终，在我们的建议下，这个电信公司重新进行了硬件选型，并基于新硬件重新进行了物理设计。本章将以这个电信公司为例，详细阐述硬件选型原则和物理设计技巧。

3.1　从数据库硬件选型谈起

在数据库系统建设初期，系统架构师需要根据当前的业务情况、对未来发展的规划以及建设经费等情况进行评估，这样可以为系统选型做好准备。以老赵所在的电信公司硬件重新选型为例，他们在 DB2 数据库服务器的选型上，综合考虑了数据处理能力、可靠性和投资成本等几个方面的需求，并结合公司 3~5 年的业务发展预期，进行系统评估和决策。

实际上，数据库硬件选型需要综合考虑多种因素，接下来从主机、存储和网络环境三个方面具体阐述。

3.1.1　对主机的考虑

经过第 1 章的学习，相信大部分读者已经了解到一个业务系统最重要的性能指标是响应时间，这是因为对最终用户来说，最敏感的就是响应时间。所以，选择什么样的主机，就是从响应时间这个角度出发考虑的。

比较：X86 服务器和 Power 服务器

在选择服务器的时候，很多客户经常问的问题就是，到底是选择 X86 服务器还是 Power 服务器？

其实，这个问题要根据具体需求来定，不能一概而论，读者可以参考下面的原则。

- 关键业务系统优先选择 Power 服务器。
- 从性价比考虑可以选择 X86 服务器。
- 由于预算的原因，可以先选择 X86 服务器作为过渡，随着数据量和用户量的变大，到了一定阶段再迁移到 Power 服务器上。

选择主机的第一步是决定 CPU 的内核个数，这要根据 OLTP 和 OLAP 不同的应用类型分别加以考虑。

针对 OLTP 系统，可以基于现有的 DB2 数据库系统来估算新系统对 CPU 的需求。例如：新系统要处理比原系统多 50%的用户请求，而 SQL 语句的复杂度和现有系统相似或更复杂，则可估算出新系统需要增加至少 50%左右 CPU 的处理能力。

针对 OLAP 系统，还需要考虑应用系统 SQL 语句的复杂程度、SQL 查询表关联情况，聚集计算等也都在考虑之内。

除了 CPU 外，还要求服务器配置足够多的内存，以及足够带宽的存储设备，以使得 I/O 方面没有瓶颈，实现高吞吐量。根据经验，对于内存，推荐 1 个 Power 处理器内核配备 8GB 内存，一个 X86 处理器内核配备 4GB 内存；对于存储，1 个 Power 处理器内核可以处理 300~500GB 活动数据，1 个 X86 处理器内核可以处理 200~300GB 活动数据。

3.1.2　对存储的考虑

在数据库系统中，出于对存储可靠性和多块磁盘 I/O 吞吐能力更强的考虑，通常使用磁盘

阵列设备。

选择磁盘阵列设备时，需要考虑业务的性能指标，主要包括 IOPS 和 TPS。IOPS 用于描述 I/O 吞吐量，用每秒处理的 I/O 访问次数来衡量。TPS 用于描述事务吞吐量，用每秒处理的事务数来衡量。

为了满足 IOPS 和 TPS 这两个性能指标，需要选择合理的 RAID（Redundant Array of Independent Disks）级别，并按照选定的 RAID 级别定义好合理的物理磁盘个数。

可以使用条带化技术将 I/O 负载均衡分布在这些物理磁盘上，也就是说将一次逻辑 I/O 请求分割成多个小部分，每个小部分同时访问存储在不同物理磁盘上的数据，这样可以最大程度地降低磁盘冲突，提升 I/O 并行度，从而提升 I/O 性能。

RAID 10 和 RAID 5 因各自实现的技术特点，在读与写的性能上又有不同。通常，RAID 10 适合写操作，而 RAID 5 更适合读操作。出于对性能的考量，推荐将数据库事务日志部署在 RAID 10 磁盘组上。

存储的性能指标与 RAID 的物理磁盘个数息息相关，那么选择多少个物理磁盘比较合适呢？根据每个处理器内核可以分担的活动数据量来计算需要的磁盘个数。

也许用户还有一个顾虑，数据库服务器与存储通讯会不会有瓶颈？鉴于这个问题，推荐采用 SAN 网络来提升通讯效率。

3.1.3 对网络环境的考虑

网络环境，包括数据库系统内部使用的网络环境和外部应用访问数据库的网络环境。

1. 数据库系统内部使用的网络环境

该环境是指 DPF 或者 pureScale 等数据库成员之间的通讯所要求的网络环境。DPF 多分区数据库的各个分区拥有自己的资源，包括 CPU、内存、磁盘、各自的数据索引配置文件和事务日志，各分区之间是并行的，因此各分区之间的数据交互，必须依赖高速网络才能保证良好的性能。

对于 DB2 pureScale 数据库而言，它的特点是各成员和 CF 之间交互频繁，采用了 RDMA 技术，直接读或写另一台计算机的内存，因此超高速 InfiniBand 网络的使用是 pureScale 高性能的保证。

2. 外部应用访问数据的网络环境

该环境是指中间件或者应用与数据库之间的网络环境。例如常见的 IBM 中间件 WAS（Web Application Server）与 DB2 数据库之间。如果使用独立的高速网络，将会大幅提升部署在 WAS 上的应用系统访问数据的性能。

3.1.4 电信公司选型结果

这个电信公司的应用特点是以 OLTP 事务型为主，以 OLAP 分析型为辅。应用的性能要求是，平时要求事务吞吐量达到 300TPS，每个事务的响应时间要求在 3s 以内完成，IOPS 要求 200 个；忙时要求事务吞吐量达到 1000TPS，每个事务的响应时间要求在 5s 以内完成，IOPS 要求 300 个。

针对上面的性能要求，首先对主机实现“鸟枪换炮”，放弃了 x3650 服务器，选用了两台

Power 740，一台用于 DB2 主机，另外一台用于 DB2 备机。每台 Power 740 都有 8 个内核，根据 1 个 CPU 内核使用 8GB 内存的原则，所以内存配置为 64GB。根据 1 个 CPU 内核分担 300~500GB 活动的原始数据，所以选择了拥有 32 块磁盘的 DS8300 存储设备，单块磁盘容量为 300GB。网络有两种，一种是外部网络，用于连接数据库服务器和应用服务器，另外一种是内部网络，用于连接 DB2 主机和 DB2 备机。详细清单如表 3-1 所示。

表 3-1　硬件选型结果

主机	两台 Power 740
操作系统	AIX 6100-06
CPU	8 核
内存	64GB
存储	DS8300，总共 24 块物理盘，单盘容量为 300GB，规划为 RAID 10。其中 8 块盘划分一个 LUN 用于存放日志，总容量为 8/2×300GB=1200GB；12 块盘划分一个 LUN，用于存放数据和索引，总容量为 12/2×300GB=1800GB；将剩余 4 块盘化为一个 LUN 用于存放临时数据和文件，总容量为 4/2×300GB=600GB
外部网络（数据库服务器和应用服务器之间）	千兆网
内部网络（DB2 主机和 DB2 备机之间）	千兆网

完成硬件选型后，接下来就需要规划数据库服务器的部署了。如图 3-1 所示，数据服务器、网络、存储和应用服务器的架构图。其中，为了实现 DB2 主机和 DB2 备机之间的高可用性，选择了 AIX 平台上的 HACMP 协议。正常情况下，DB2 主机负责处理应用服务器的请求，当主机出现故障时，HACMP 协议可以实现备机对主机的接管，从而满足业务对数据库不间断运行的需要。

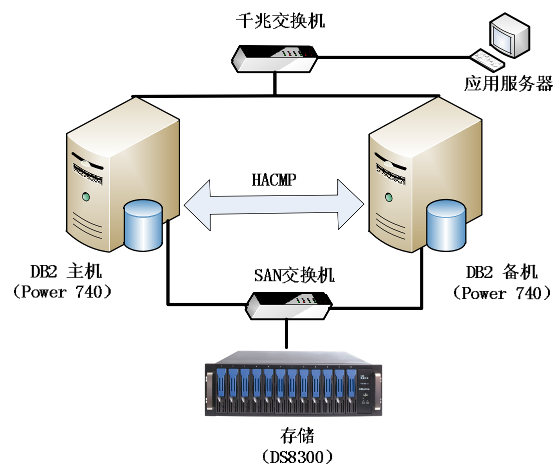


图 3-1　硬件架构图

3.2　存储设计

本文对主机、网络环境不再赘述，着重谈谈存储系统的设计。

数据库和存储系统息息相关。如果为数据库选择硬件，第一要考虑的就是存储架构。你会

发现，DB2 所使用的存储越快，则 I/O 性能将越好。

存储是存放数据和日志的地方，因此存储的设计显得格外重要。特别是随着现代服务器技术的发展，磁盘技术的发展远远落后于 CPU 和内存的技术发展，这个时候如果出现了存储设计不合理，那么由此导致的 I/O 瓶颈往往会成为性能最薄弱的地方。

小故事：网页设计人员替我背黑锅了

10 年前，我还在北京读研究生，由于家境贫寒，想赚点学费，所以利用暑假时间帮一家电信服务商建设和维护数据库。刚上线的时候，数据库运行正常，但是随着数据量和用户量的快速增长，系统性能问题开始出现，而且也频频出现故障。我那时白天非常繁忙，到了晚上过的也很充实，因为做梦都是性能调优和故障诊断。

但是，有一天发生的事让我羞愧不堪。那天，这家电信服务商的 CEO 带着风险投资人从上海来到北京，开始用微软公司的 PowerPoint 软件讲述着公司的美好未来，这时风险投资人主动提出希望看看这个系统，于是 CEO 点击一个页面又一个页面进行演示。当他点击到一个关键页面的时候，突然系统就没有任何反应了，搞的大家非常尴尬，估计风险投资人也没有兴趣了。会后，这个 CEO 感觉丢了大面子，于是他把网页设计人员找来，大声痛骂一番。

事后我才知道，那是存储设计不当引起的数据库性能不佳导致的，与网页设计人员没有半点关系。从此以后，我痛定思痛，告诉自己一定要搞好性能优化工作，不然让他人轻则被骂，重则失去重大投资机会啊。

3.2.1 磁盘与磁盘阵列技术

磁盘处于整个存储系统的最底层，核心的业务数据通常都存放在磁盘上。由于任何计算都在主存或高速缓存中进行，因此对于数据库系统，最关心的问题是如何快速在磁盘和内存之间移动数据，换句话说，就是如何实现磁盘阵列的高效读写。

那么如何衡量一个磁盘的读写能力呢？有下面一些指标：

- **磁盘转速（Rotational Speed）**：决定磁盘内部传输率的关键因素之一，磁盘转速为 5400 转/分钟、7200 转/分钟和 15000 转/分钟，现在主流的都是 15000 转/分钟。理论上，转速越快越好，但是转速越快发热量就越大。
- **平均寻道时间（Average Seek Time）**：即磁盘管理器将磁头定位到合适的柱面所花费的时间，它描述磁盘读取数据的能力，单位为毫秒。
- **平均延迟时间（Average Latency Time）**：当磁头移动到数据所在的磁道后，然后等待所要的数据块继续转动到磁头下所用的时间。
- **平均访问时间（Average Access Time）**：即磁头找到指定数据的平均时间，通常是平均寻道时间和平均延迟时间之和。

注意：缓存大小对存储读写性能影响很大

对于读者来说，如果读取的数据都能在缓存中命中的话，将大大减少磁盘寻道所需要的时间。对于写来说，一般存储阵列只要求数据写到缓存就算完成了写操作，当写入的数据积累到一定程度，存储阵列才会把数据刷新到磁盘，这样可以实现批量写入。所以，缓存大小对存储读写性能影响很大。

通常上述指标是由磁盘本身决定的，有时候，单块磁盘的 I/O 能力无法满足应用的性能需要。所以，可以考虑将多个相对比较廉价的磁盘组合起来，并相互连接，以组成一个磁盘组，从而使其性能和容量达到或超过一个价格更昂贵的单个大型磁盘。RAID 正是基于多块廉价磁盘组合而出现的，它推出了一系列的级别，包括 RAID 0、RAID 1、RAID 5、RAID 10 等，如图 3-2 所示，由 5 块磁盘组成的磁盘阵列以及相应的 RAID 0、RAID 1 和 RAID 5 级别。

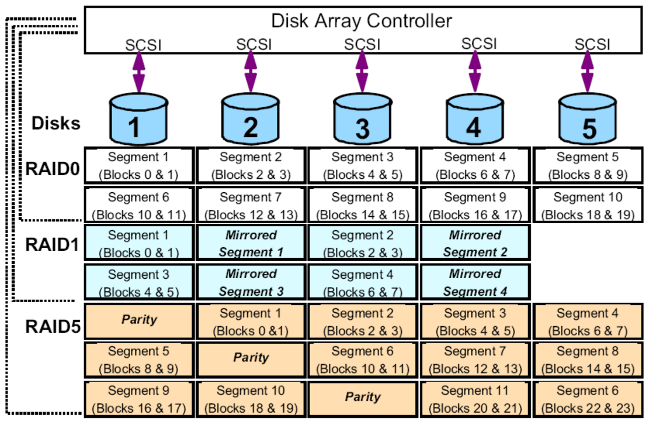


图 3-2　磁盘阵列示意图

1. RAID 0

RAID 0 方案中，采用条带化，把数据分割成多个数据块，分别写入不同的物理磁盘，在多个磁盘上均衡了 I/O 负载，提升了 I/O 并行度，从而提升了效率。需要注意的是，该方案牺牲了可靠性，达到了 I/O 的最高性能。

RAID 0 的特性：磁盘的有效空间利用率为 100%，但不具有数据保护功能，不适合于关键数据。在实践工作中，实施 RAID 0 最少需要两块硬盘（理论上一块盘也可以创建 RAID 0）。

2. RAID 1

RAID 1 不仅保存数据，还将同一份数据保存在两个不同的磁盘上以提升可靠性。以图 3-2 为例，1 号磁盘和 2 号磁盘互为镜像，3 号磁盘和 4 号磁盘互为镜像。

RAID 1 的特性：通过磁盘镜像，将同样的数据复制存储到不同的磁盘上，从而提高了数据的高可靠性。读写操作效率较高，但是实施成本高。

3. RAID 5 级

RAID 5 在所有的磁盘上分布校验块用于保持数据的完整性，并在磁盘出现故障时进行重建。如果阵列内的某个磁盘出现故障，丢失的数据可以根据其他磁盘上的校验块进行重建。这种分布有两个优点：首先，几个写操作可以并行处理，因为消除了单一校验盘的瓶颈；第二，读请求有很高的并行度。因为数据分布于所有的磁盘，读请求可以操作所有的磁盘，而在单一校验盘的系统中，校验盘从不参加读操作。

RAID 5 的特性：更有效地利用所有冗余 RAID 配置的磁盘容量，保持了良好的读写性能。不过，磁盘故障会影响吞吐速率，故障后重建信息的时间比镜像配置情况下要长。对于各类读操作和大的写操作，RAID 5 与其他带有冗余的 RAID 系统相比具有最好的性能；而对于小的写

操作，它没有 RAID 1 的性能好。

比较：RAID 6 和 RAID 5 有什么区别

RAID 6 是在 RAID 5 基础上，为了进一步加强数据保护而设计的一种 RAID 方式，实际上是一种 RAID 5 扩展。与 RAID 5 的不同之处在于除了每个硬盘上都有同级数据的校验区外，还有一个针对每个数据块的校验区。这样一来，等于有了两个校验保护屏障，因此 RAID 6 的安全性更好。但是，由于增加了一个额外的校验，所以读写性能较 RAID 5 差一点，另外也减少了有效存储空间。

4. RAID 10 级

RAID 10，分为 raid 1+0 和 raid 0+1 两种类型，它综合了条带（Striping）和镜像（Mirroring）技术。RAID 1+0 首先创建 2 个独立的 Raid 1，然后将这两个独立的 Raid 1 组成一个 Raid 0，数据被有序的写入两个 Raid 1 中；RAID 0+1 是先做 RAID 0 条带，再做 RAID 1 镜像，这样写入速度快，读的速度和 RAID 1+0 一样。

RAID 10 的特性：RAID 10 是 RAID 1 和 RAID 0 的结合，它的优点是同时拥有 RAID 0 的超凡速度和 RAID 1 的数据高可靠性。此配置要求至少 4 块硬盘，在所有 RAID 等级中，性能和保护功能都是最佳的。

常用 RAID 级别特点总结见表 3-2。读写性能是以各 RAID 级别比较得出的结论；需要的磁盘个数是技术上可以支持的个数，例如 RAID 0 可以只使用一块磁盘，但是并不具有实际意义。

表 3-2　RAID 级别总结

RAID 级别	RAID 0	RAID 1	RAID 10	RAID 5
别名	条带	镜像	镜像阵列条带	分布奇偶位条带
容错性	无	有	有	有
冗余类型	无	复制	复制	奇偶校验
读性能	最高	低	高	高
写性能	最高	低	高	较高
需要的磁盘数	一块或多块	两个或 2N 块（偶数个盘）	四块或 4N 块	三块或三块以上
可用容量	磁盘总容量	磁盘总容量的 50%	磁盘总容量的 50%	（N-1）/N 个磁盘的总容量
典型应用场景	提供最好的性能，但是安全性最差，实际生产中用的不多	安全性要求高的交易系统	安全性要求高的交易系统	安全性要求不高的数据仓库或交易系统

3.2.2　条带化

在上节中，多次提到一个词“条带化”。实际上，现代存储阵列、现代操作系统、各种第三方软件都支持条带化技术，例如 IBM 存储 DS8300 就支持条带化，AIX、HP-UX 等都在 LVM（逻辑卷管理）上支持条带化，Oracle 通过 ASM 支持条带化，DB2 通过自动存储管理有限支持条带化。这一节专门来谈谈条带化。

很多 DBA 都遇到过磁盘冲突，磁盘冲突是指多个进程同时访问同一块磁盘，这是 I/O 设计中需要重点考虑的问题。由于 I/O 经常是 DB2 的性能瓶颈，因此最好的解决方法就是使 I/O 并行，这就需要将数据分布在多个磁盘上以共同分担 I/O。

条带化技术的出现很好的解决了上述问题，它会将数据均匀地分布到不同的磁盘上，同时也将负载分担到多个磁盘上。这样，多个磁盘就可以并行工作，吞吐量会大大提高。通过条带化，DBA 可以很轻松地做到 I/O 负载均衡而无须去手工配置。

那么，采用条带化技术后，数据在物理磁盘上是怎样分布的？和未条带化有什么区别？

图 3-3 描述的是一个未经条带化处理的连续数据的分布，所有的数据会依次写满三个磁盘，先写满第一个，然后再写第二个、第三个。

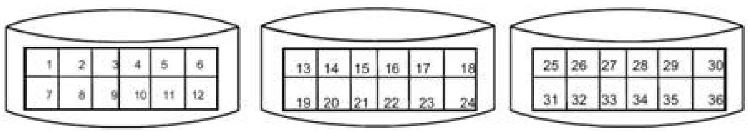


图 3-3　未经条带化处理的连续数据

图 3-4 描述的是一个已经条带化处理的连续数据的分布，它以扩展分区（Extent）为单位，依次写入三个磁盘，然后不断循环往复。

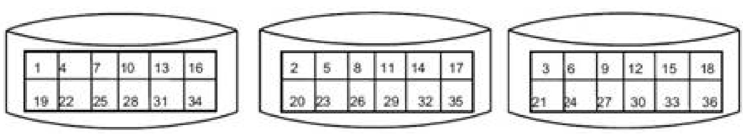


图 3-4　已经被条带化处理的连续数据

可以发现图 3-4 中对连续数据的读写有更大的并发能力，如果用户需要读取 1 到 3 扩展分区中的数据时，如果没有条带化，只能从第一块磁盘中读取；如果经过条带化，那么就可以同时访问三个磁盘，条带化磁盘的速度是未经条带化磁盘的三倍。同样，如果写入大量数据，在图 3-3 中，只能操作一个物理磁盘进行写操作，而图 3-4 中，可以同时对三块物理磁盘进行写操作。

当使用条带化时，决定条带化效果的因素是条带深度（stripe depth）和条带宽度（stripe width）。条带深度指的是条带块的大小，也叫条带单元；条带宽度指的是一个条带分布的驱动数。设计人员需要根据系统的 I/O 要求来合理的选择这些设置。

1. 条带深度

条带的深度指的是条带的大小，也叫条带单元。条带化的初衷是为了提高 I/O 并行性，也就是说希望每次 I/O 操作，都能得到最大程度的并行。

条带的配置和系统 I/O 的并发度有关，在高并发且逻辑 I/O 请求的大小都比较小的情况下，希望一块磁盘能同时响应多个 I/O 请求，这时就需要考虑选择比较大的条带深度。在高并发系

统中，条带深度为（n×page size），其中 n 为大于 1 的整数。通过粗粒度条带能实现最大的 I/O 吞吐量。

在那些存在大的逻辑 I/O 请求的低并发系统中，需要多块磁盘同时响应一个 I/O 请求。由于 I/O 请求一般是序列化的，为了避免出现 I/O 集中的热点磁盘，需要避免逻辑 I/O 只被一块磁盘处理。这时应该选择小的条带深度，使一个逻辑 I/O 分布到多个磁盘上，从而实现 I/O 的负载均衡，这就叫细粒度条带。条带深度的大小必须为（n×page size），其中 n 为小于预取参数（prefetch size）大小的整数。

比较：逻辑 I/O 与物理 I/O

逻辑 I/O，是操作系统发起的 I/O，目标数据可能会在磁盘上，也可能会在内存中。

物理 I/O，是设备驱动发起的 I/O，从磁盘读取目标数据。

逻辑 I/O 会转化为物理 I/O 请求。在条带化的概念中，一个逻辑 I/O 会转化成若干个物理 I/O，多个逻辑 I/O 也可以通过一次大的物理 I/O 完成。

2．条带宽度

条带宽度指的是条带深度的产量或者一个条带集中的驱动数。无论是一个还是多个磁盘响应一个逻辑 I/O，都希望物理设备只处理一次 I/O。因而在确定了条带深度的基础上，需要保证：

```
条带宽度 >=  I/O 请求的大小 ÷ 条带深度。
```

在 DB2 数据库级别影响单次物理 I/O 最重要的因素就是预取大小。为了提高 I/O 的性能，在设置 DB2 预取大小时，必须使其与条带深度和条带宽度有较好的匹配。首先要保证的就是预取的大小必须是条带深度的整数倍，其次预取大小最好是条带深度与条带宽度乘积的整数倍。这样就可以使预取时所执行的物理 I/O 被均衡地分布到所有的物理存储中。

3.2.3　存储设计

存储设计的成败，会直接影响 DB2 数据库的性能。这一节将从以下几个方面，讨论 DB2 存储设计时需要注意的一些技术要点。

1. RAID 级别的选取和设计

（1）对于数据和日志，应选择不同的 RAID 组，建议数据采用 RAID 5 或者 RAID 10，日志采用 RAID 10。

（2）一般小 I/O 的数据库类操作，OLTP 应用例如 ERP 系统，建议采用 RAID 10，而对于 OLAP 应用例如数据仓库或者商业智能系统，建议采用 RAID 5。

小技巧：判断 OLTP 还是 OLAP

可以通过下面的方法区分应用系统属于哪种类型。如果遵循 70/30/50 原则，即 70%读、30%写、Cache 命中率 50%，这就是典型的高并发的 OLTP 系统。如果系统是 100%读，则这是典型的 OLAP、DSS 系统。

2. 条带化的使用

（1）要考虑 DB2 系统的 I/O 特点，以此来选择存储上的条带化大小(stripe size)。对于 OLTP 应用，数据库 I/O 基本是随机小块读写，所以选择较大的条带大小可以调高 I/O 吞吐能力；而对于 OLAP 应用，数据库 I/O 是连续大块读写，所以选择较小的条带大小避免 I/O 请求局限在一块磁盘上。

（2）确保操作系统层面和存储层面的条带化大小设置一致。

（3）最多只能采用两次条带化，条带化并不是越多越好。存储设备是通过 RAID 实现条带化的，而 DB2 是通过表空间容器实现条带化。从最佳实践的角度出发，建议条带化不要超过两次。

3. DB2 自动存储管理技术

DB2 自动存储管理（Automatic Storage Management）技术是为数据库配置存储的一种简单而有效的方式。例如创建数据库 MYDB 的 CREATE DATABASE 语句：

```
CREATE DATABASE MYDB ON /data1, /data2, /data3  DBPATH ON /mydbpath
```

上面的命令有 3 个存储路径，分别为 data1、data2 和 data3。每个存储路径都是一个单独的文件系统，每个文件系统都是通过专用的 LUN 创建的。DBPATH 参数被设为另一个单独的文件系统/mydbpath，事务日志和 DB2 元数据都存放在这个文件系统上。

> **注意：CREATE DATABASE 语句**
>
> 默认的情况下，在使用 CREATE DATABASE 命令创建数据库时将启用自动存储管理。

DB2 存储管理器使用简单的条带化方法，可确保并行性，不至于使某个存储路径过早地被填满。

当需要为数据库增加空间时，应尽量均衡扩展所有已有的路径。也就是说，等量地增加每个文件系统的容量。例如，为了给数据库 MYDB 增加空间，最佳选择是等量增加文件系统/data1、/data2 和/data3 的容量。

如果不能均衡扩展存储路径，那么使用 ALTER DATABASE 命令的 ADD STORAGE ON 子句增加一组新的存储路径。这组新的存储路径应该与原有存储路径有相同的存储容量，例如下面的命令新增加一组存储路径，分别为 /data4、/data5 和 /data6。

```
ALTER DATABASE ADD STORAGE ON /data4, /data5, /data6
```

4. 与存储相关的 DB2 配置参数

（1）DB2_PARALLEL_IO 注册变量

DB2_PARALLEL_IO 允许用户指定受影响的表空间和每个容器的并行度，即每个容器下面的磁盘数目。一个合理的 DB2_PARALLEL_IO 设置如下：

```
db2set DB2_PARALLEL_IO= *:4
```

“*”通配符告诉 DB2 管理器应用这个设置到所有的表空间，而且每个容器的磁盘数是 4。

（2）使用 NO FILE SYSTEM CACHING 子句

在一些情形下，DB2 可从文件系统缓存的使用中获益，特别是在要频繁查询 LOB 数据表的情形中。因为这些 LOB 数据不会写入 DB2 缓冲池，所以使用文件系统缓存性能会有助于 I/O 读取的性能。

但是，在很多场景下，使用文件系统缓存并不会带来性能的提升，所以需要将文件系统缓存关闭，这可以通过在 CREATE TABLESPACE 和 ALTER TABLESPACE 命令使用 NO FILE SYSTEM CACHING 子句来实现。

NO FILE SYSTEM CACHING 子句用于启用非缓冲 I/O，从而禁用特定表空间的文件高速缓存，这样数据库管理器就会自动使用直接 I/O 或者并发 I/O。

比较：直接 I/O 与并发 I/O

直接 I/O（DIO），通过直接将数据从磁盘复制到合适的缓冲区，消除了与缓存 I/O 关联的双重缓冲。但是，直接 I/O 执行了写入序列化和 Inode 锁定来维护数据完整性。

并发 I/O（CIO），提供了对数据的直接访问能力，无须额外的缓冲和锁定，可以允许多个线程并发地读写相同文件中的数据。

---

5. 日志和数据分开

DB2 事务在进行增删改的时候需要先写日志才能提交，这意味着日志的写入速度对性能有很大影响。因此，要把日志盘从物理上单独分出来，最好规划成 RAID 10，给予最高优先级，防止数据读写与日志写之间竞争磁头。

与日志盘不同，数据盘的容量需求比较大，对于读的要求更高一些，再加上 DB2 的数据写都是异步的，所以对写入速度要求高的场景下选择 RAID 10，否则选择 RAID 5。

6. 尽量发挥文件系统的 CIO 和 DIO 能力，减少裸设备的使用

裸设备与文件系统的最大不同在于它可以避开文件系统的缓存以及锁机制，因此，可以通过裸设备提高数据库的性能。但是，裸设备的弊端是可管理性差，容易出问题，而且不被 DB2 的自动存储管理特性所支持。

随着文件系统的技术发展，现在许多文件系统，例如 AIX 下面的 JFS2 都支持 CIO 模式，也就是说可以利用文件系统的 CIO 或 DIO 能力达到与裸设备相似的性能，而且还兼顾了可管理性和安全性。

开启 CIO/DIO 有两种方法：第一种方法是将文件系统 mount 成 CIO 或 DIO，这样做会导致被 mount 的文件系统上所有文件都使用 CIO 或 DIO；第二种方法在创建或修改表空间时指定 NO FILE SYSTEM CACHING 属性，它只是对表空间所对应的容器使用 CIO/DIO，并不会影响到文件系统上的其他文件。

无论使用上述哪种方法开启 CIO 或 DIO，都必须在操作系统中正确地安装和配置 AIO 库。AIO 库在 Linux 平台和 AIX 平台的安装、配置方法是有区别的，在 Linux 平台使用 CIO 或 DIO 需要检查 AIO 库是否安装，检查方法如下：

```
rpm -q libaio
libaio-0.3.106-3.2
libaio-0.3.106-3.2
```

如果 aio 库已经安装，下一步需要在/etc/sysctl.conf 文件中配置 AIO 的内核参数，并执行 sysctl -p 命令让配置生效。

```
fs.aio-max-nr = 1048576
```

关于 AIX 系统中如何配置 AIO，请读者参考本书第 11 章 11.3.4 小节有关 IOCP 的内容。

3.2.4　为电信公司规划存储

从 3.1.4 小节了解到，这个电信公司最终采用了 DS8300 存储阵列，总共配置 24 块物理盘，单盘容量 300GB，最终选择 RAID 10 技术。首先，在数据高可靠性方面，RAID 10 技术可以满足公司对数据高可靠性的要求；其次，在读写性能方面，RAID 10 技术提供较高的读写性能可以满足应用系统的要求；最后，在磁盘容量方面，可以满足公司未来 5 年的发展需要。

具体划分如下：其中 8 块盘划分一个 LUN 用于存放日志，总容量为 8/2×300GB=1200GB；12 块盘划分一个 LUN，用于存放数据和索引，总容量为 12/2×300GB=1800GB；将剩余 4 块盘化为一个 LUN 用于存放临时数据和文件，总容量为 4/2×300GB=600GB。

另外，考虑到裸设备难于管理，而且性能也未必比文件系统好，所以最终选择文件系统。最终存储与文件系统设计如表 3-3 所示。

表 3-3　存储与文件系统设计

文件系统	名称	RAID 级别	物理盘个数	可用容量	用途
/db2log	LUN1	RAID 10	8	1200GB	存放日志
/db2data	LUN2	RAID 10	12	1800GB	存放数据
/db2index	LUN2	RAID 10	12	1800GB	存放索引
/db2temp	LUN3	RAID 10	4	600GB	存放临时数据

测试存储的实际读写速度，是存储设计阶段必须要做的事情，这样做的好处是：第一，可以验证是否能满足数据库系统的 I/O 要求；第二，可以对当前存储器 I/O 性能有准确的了解，从而为后期性能调优做参考。

下面使用 dd 命令测试/db2data 文件系统的读写能力，这里的文件系统/db2data 实际对应着一个存储器上划分的 LUN，测试结果表明可以满足 I/O 需要。

```
[root@db2 ~]# dd if=/dev/zero of=/db2data/test.file bs=8192 count=5000000
5000000+0 records in
5000000+0 records out
40960000000 bytes (41 GB) copied, 179.411 seconds, 228 MB/s
```

3.2.5　为电信公司创建数据库

上一节完成存储设计后，接下来就可以创建数据库了，语句如下所示：

```
create database crmdb automatic storage yes on /db2data dbpath on /db2log ALIAS cdb
```

```
using codeset UTF-8 territory CN pagesize 8 k;
```

这个数据库的名字为 crmdb，以表示这是客户关系管理系统数据库，下面介绍一下上面命令中其他选项的具体含义，

- automatic storage yes：表示使用自动存储管理功能，其中自动存储路径“/”为 db2data。关于自动存储表空间的具体介绍，请阅读 3.3.1 节有关内容。
- dbpath on /db2log：指定数据库 crmdb 的事务日志和元数据存储路径，当然也可以随时通过修改数据库配置参数 NEWLOGPATH 来调整。
- ALIAS cdb：指定 crmdb 的别名为 cdb。
- codeset UTF-8：指定 crmdb 数据库使用 UTF-8 编码，这样可以存储各种语言的内容。
- territory CN：它决定数据库中的时间和日期格式为中文。
- pagesize 8 k：指定数据库的默认页大小为 8KB，为什么要选择 8KB，请阅读 3.3.4 节有关内容。

3.3 表空间设计

表空间是数据的载体，也是数据库逻辑层与物理层之间的接口，表空间的规划和管理与底层存储的规划和管理密不可分，在这一节，将深入讲解 DB2 表空间有关内容。

3.3.1 表空间管理方式

表空间是数据库中的一个存储结构，表、索引和大对象数据（LOB 和 LONG）都被创建在表空间中。作为应用开发人员，只需要了解到表空间一层，不需要关心表空间的物理存储结构。作为运维人员，需要深入了解表空间的物理组织结构。

我们在技术交流的时候，发现很多 DBA 并不熟悉 DB2 表空间的管理方式，接下来深入探讨一下。

1. 系统管理表空间（SMS）

系统管理表空间是指在创建表空间时指定 MANAGED BY SYSTEM 选项，并且指定一个或若干操作系统路径。容器会被创建在这些路径当中，一个容器文件中只包含一个对象的数据，因此有多少对象就会有多少个容器文件。容器不事先分配，而是随着数据的插入或删除，容器文件的大小自动调整。

例如，下面的语句创建了系统管理表空间 SMSTS：

```
CREATE TABLESPACE SMSTS
     MANAGED BY SYSTEM
     USING ('/path1', '/path2', '/path3')
```

在创建表空间时可以指定若干路径，表空间会以对象为单位在这些路径上创建文件，一个文件中只有一个对象的数据，其物理存储结构如图 3-5 所示。如果在表空间上创建了表 A 并插

入了若干记录，那么在路径 1、路径 2 和路径 3 下面都会出现文件 SQL00001.DAT。如果再创建另一个表，会出现另一个文件 SQL00002.DAT，一般每个数据库实体对象（比如表或索引）对应一个文件名。

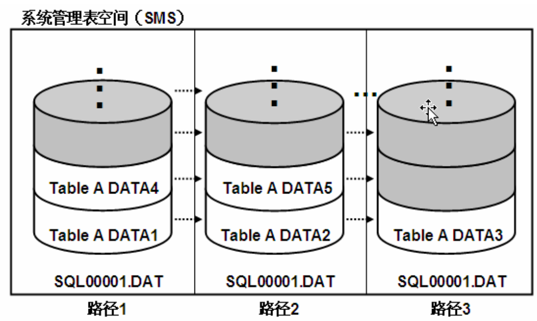


图 3-5　SMS 表空间容器的存储结构

那么 SMS 表空间读写机制是什么呢？在 DB2 中，采取轮循（Round Robin）的方式。也就是说，当一个扩展块的数据写到路径 1 下面的文件后，接下来一个扩展块的数据写到路径 2 下面的文件中，再接着一个扩展块数据写到路径 3 下面的文件中，最后，又会翻回来写路径 1，如此反复。

采用系统管理表空间，用户无须自己管理，一切由系统进行打理，好处是比较简单，管理方便，但是性能上不一定是最佳的。一般系统临时表空间会使用系统管理表空间。

2. 数据库管理表空间（DMS）

在创建由数据库管理的表空间时，需要指定具体的文件或裸设备作为容器，表空间的所有对象都被放在这些容器中，一个文件里面可以有不同对象的数据。文件的存储空间会预先分配好，在预分配的存储空间用完以后可以手动或者自动追加空间。

例如，下面的语句创建了数据库管理表空间 DMSTS：

```
CREATE TABLESPACE DMSTS
     MANAGED BY DATABASE
     USING (FILE '/path1/dmsdata1' 5000,
            FILE '/path2/dmsdata2' 5000,
            FILE '/path3/dmsdata3' 5000)
```

数据库管理表空间的容器与 SMS 表空间不同，容器是事先分配的而不是随需分配的，其物理存储结构如图 3-6 所示。从图 3-6 可以看出，DMS 表空间的数据读写也采用了轮循方式，即数据从第一个容器开始写入，写完一个扩展块写第二个容器的扩展块，依次类推，最后再返回写第一个。

采用数据库管理表空间，好处是性能比较高效，但是管理上稍微复杂一点，下面具体来讲。

（1）改变数据库管理表空间（DMS）容器的大小

下面的例子将表空间 DMSTS 的三个容器重设为 2000 个页大小，RESIZE 既可以把容器加大也可以把容器缩小。

```
ALTER TABLESPACE DMSTS
    RESIZE （FILE '/path1/dmsdata1' 2000,
            FILE '/path2/dmsdata2' 2000,
            FILE '/path3/dmsdaat3' 2000）
```

如果表空间的属性 AUTORESIZE 被设为 YES，则不需要手动管理表空间容器的大小，只需要保证相应的操作系统还有足够的空间即可。但 AUTORESIZE 只能加大容器，要缩小容器还需要手动执行 ALTER TABLE…RESIZE 操作。

管理表空间容器尺寸的最佳实践是保证容器的扩展是等大小的，这样可以防止热区的出现。所谓热区，指的是表空间的某一个或若干容器比其他容器大，导致某一块磁盘访问频率明显高于其他磁盘。

通常在规划数据库存储时，会把不同的容器放到不同的文件系统中，它们有自己独立的磁头，这样可以最大限度地利用并行 I/O。如图 3-7 所示，在读扩展块 4 的同时可以读扩展块 5 和 6。如果某一个容器比其他容器大，在其他容器空间用尽后会集中使用一个容器，这样扩展块 10~12 不能同时被并行访问，并且导致某个磁盘访问量大于其他磁盘，产生读写冲突，影响读写性能，因此在规划中要尽量避免这种热区。

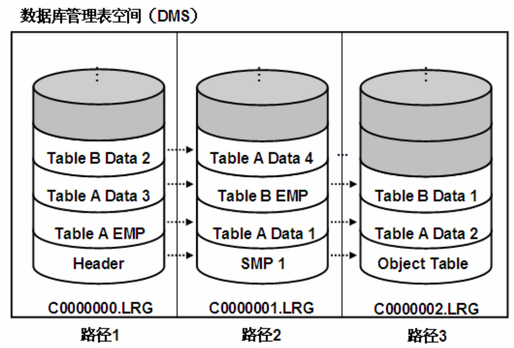


图 3-6　DMS 表空间容器的存储结构

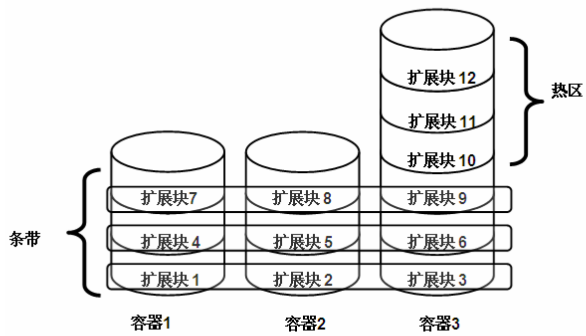


图 3-7　DMS 表空间容器的热区

（2）为数据库管理表空间（DMS）增加容器

在某些情况下，原有的磁盘已经没有空间，这时就需要为表空间增加容器，把容器放到其他的空闲磁盘上。如果为数据库管理表空间增加容器，会导致数据自动重新分布。

下面例子中，为表空间 DMSTS 增加两个 2000 个页大小的容器。

```
ALTER TABLESPACE DMSTS
    ADD （FILE '/path4/dmsdata4' 2000, FILE '/path5/dmsdata5' 2000）
```

如图 3-8 所示，初始阶段有两个容器 C0 和 C1，数据为 t0，当增加两个容器 C2 和 C3 时，会自动触发数据重新平衡，t0 的数据被平均分布到了容器 C0、C1、C2 和 C3。所以使用这个命令要小心，最好是在工作负载比较低的时间进行，否则数据重新分布会对系统性能产生影响。

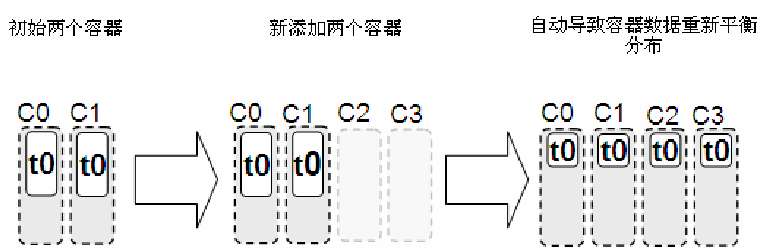


图 3-8　DMS 表空间添加容器后的过程

（3）为数据库管理表空间（DMS）增加条带集（stripe set）

如果在增加容器的时候，不希望立刻进行数据自动重新分布，这可以通过为表空间增加条带集（stripe set）来实现。这样只有原有的容器用完了，才会用新的容器。下面的命令为 DMSTS 表空间增加了一个新的 stripe set，其中包含两个容器文件，初始大小都为 2000 页。

```
ALTER TABLESPACE DMSTS begin new stripe set (
    file '/path4/dmsdata4' 2000,
    file '/path5/dmsdata5' 2000)
```

3. 自动存储表空间（Automatic Storage）

在 DB2 中，经常会提到自动存储表空间。所谓自动存储并不是一种新的表空间管理形式，它实际上还是使用 DMS，只不过在创建表空间时不需要指定任何路径或者文件，DB2 会在指定的路径创建容器，并且保证数据可以均匀地分布在这些容器中。

下面的语句，给数据库增加了三个存储路径，分别为/path1、/path2 和/path3：

```
ALTER DATABASE ADD STORAGE ON '/path1', '/path2', '/path3'
```

为自动存储管理的数据库添加路径以后，并不会马上利用到这些存储路径，直到原有路径的空间用尽，这点与 DMS 正好相反。

如图 3-9 所示，初始阶段有两个路径 C0 和 C1，数据为 t0，添加两个路径 C2 和 C3 后，并不会触发数据重新分布，接下来的数据 t1 会继续使用路径 C0 和 C1 直到 C0 和 C1 用完为止，这时再到来的数据 t2 就会被放在新的路径 C2 和 C3 了。

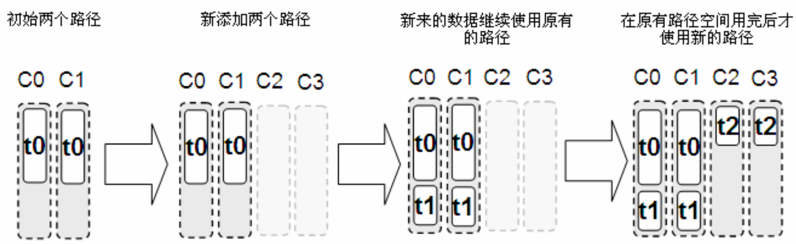


图 3-9　为自动存储管理数据库添加存储路径

3.3.2　表空间类型

上一节介绍了表空间的管理方式，本节将介绍 DB2 中不同的表空间类型，根据用途可以分为以下 5 种。

（1）系统编目表空间

在创建数据库之后，DB2 默认会自动创建，名字为 SYSCATSPACE，用来存放表的定义、授权和统计信息等系统数据。

（2）系统临时表空间

默认创建，名字为 TEMPSPACE1，用于存储 SQL 操作期间包括排序、表连接等所需要的内部临时数据。

（3）用户临时表空间

默.认不创建，用于存储临时表。

（4）常规表空间

用于存放数据或索引，常规表空间只能支持单表最大尺寸为 512GB 的数据，现在不建议使用常规表空间了。

（5）大型表空间

从 DB2 V9 开始引入，从字面上可以看出大型表空间的容量要比常规表空间更大，可以支持单表最大尺寸为 16TB 的数据，建议使用。

3.3.3 表空间参数设置

使用 CREATE TABLESPACE 命令创建表空间时，可以指定下面几个参数，这几个参数对性能有重要影响，当然也可以在创建后通过 ALTER TABLESPACE 语句修改。

1. 页大小（Page Size）

DB2 支持的页大小，有 4KB、8KB、16KB 和 32KB 四种，如表 3-4 所示，不同页大小的表空间允许的单表最大尺寸是有限制的，其中，从 DB2 V9 开始支持大型表空间，其中在 32KB 页大小情况下，单表最大尺寸可以达到 16TB。

表 3-4　单表支持的最大尺寸

页大小	DB2 V8 单表最大尺寸	DB2 V9 单表最大尺寸
4KB	64GB	2TB
8KB	128GB	4TB
16KB	256GB	8TB
32KB	512GB	16TB

那么，如何选择页大小呢？这要根据不同的应用来定。针对 OLTP 事务型应用，由于以随机读写为主，采取较小的页面，例如 4KB 或 8KB 更合适。针对 OLAP 分析型应用，由于以连续大片读为主，采取较大的页面，例如 16KB 或 32KB 更合适。

2. 扩展块大小（Extent Size）

大家知道，数据是以 extent（扩展块）为单位存放的，一个 extent 由一个或多个页组成，其大小（EXTENTSIZE）在创建表空间时指定。一般 RAID 条带大小是 32KB、64KB、128KB 等，表空间的 EXTENTSIZE 应该被设置成可以包含一个 RAID 条带的页数，计算公式为：

```
EXTENTSIZE  =   Stripe Size (KB) / Page Size (KB)
```

例如，在一个 4D+1P 的 RAID 5 磁盘阵列，条带大小是 512KB。如果页大小使用 8KB，那么 EXTENTSIZE 设置为 64（512KB / 8KB ）比较合适。

3. 预取大小（Prefetch Size）

预取操作在查询使用所需的数据之前读入这些数据，因为数据已经在内存中了，这样就不必等待执行 I/O 了。

通过使用 ALTER TABLESPACE 语句可以轻易地修改预取大小，如果表空间驻留在一个磁盘阵列上，最优设置如下：

```
PREFETCH SIZE = EXTENTSIZE *(RAID 数据盘的个数)
```

例如表空间驻留在一个 4D+1P 的 RAID 5 磁盘阵列上，由于数据盘的个数为 4，那么 Prefetch Size 的设置如下：

```
PREFETCH SIZE = EXTENTSIZE * 4
```

当然，从 DB2 V9 以后，可以在创建表空间的时候，将 Prefetch Size 设置为 Automatic，这样就可以自动预取大小了。

3.3.4　为电信公司设计表空间

这个电信公司的数据库系统以事务型为主，分析型为辅。我们从上一节知道，事务型应用随机读写数据的频率非常高，通常选择较小的页大小，这样可以减少缓冲区空间；分析型应用以长事务为主，一次访问大量连续数据的频率非常高，通常选择较大的页大小，这样可以降低 I/O 请求次数。

针对这个电信公司的业务系统特点，我们对表空间做了如下设计：

（1）选择页大小大小为 8K，这样可以较好的支持事务型，也能兼顾分析型。

（2）选择扩展块大小为 32。根据 3.2.4 节的存储规划，12 块盘划分一个 LUN，组成 RAID10，用于存放数据和索引，条带大小是 512KB，那么 EXTENTSIZE 设置为 64（512KB / 8KB）。

（3）预取大小。目前使用的 DB2 版本为 DB2 V9 FP5，可以在创建表空间的时候，将 Prefetch Size 设置为 Automatic，这样就可以自动预取大小了。

（4）放弃自动存储管理，选用 DMS 表空间，这样可以灵活定义。

（5）数据和索引使用不同的表空间存储。

（6）将热度不同的表放置于不同的表空间。

（7）每个表空间使用独立的缓冲池。

根据上面的设计，一共规划 4 个大型表空间：表空间 TSDHISTORY 用于存放历史冷数据；TSDCURRENT 用于存放当前热数据；TSIHISTORY 存放历史冷数据的索引；TSICURRENT 存放当前热数据的索引；最终表空间规划情况如表 3-5 所示，其中文件系统/db2data 和/db2index 在 3.2.4 节讲过；缓冲池 tsdbhis、tsdbcur、tsibhis 和 tsibcur 将在 3.4.5 节讲解。

表 3-5　表空间规划

表空间	容器个数	文件系统	缓冲池	初始大小	自调整尺寸	用途
TSDHISTORY	16	/db2data	tsdbhis	256GB	是	历史冷数据
TSDCURRENT	8	/db2data	tsdbcur	128GB	是	当前热数据
TSIHISTORY	8	/db2index	tsibhis	128GB	是	历史冷数据索引
TSICURRENT	4	/db2index	tsibcur	64GB	是	当前热数据索引

根据表 3-5 的规划，下面是创建表空间的语句，读者需要注意，每个表空间具有不同的容器个数，每个表空间具有不同的初始化大小，但尺寸都是自调整的。

```
create large tablespace TSDHISTORY pagesize 8 K managed by database
using (
```

```
FILE  '/db2data/TSDHISTORYC001.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC002.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC003.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC004.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC005.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC006.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC007.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC008.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC009.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC010.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC011.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC012.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC013.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC014.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC015.DAT' 16 G,
FILE  '/db2data/TSDHISTORYC016.DAT' 16 G
) autoresize yes extentsize 64 prefetchsize automatic bufferpool tsdbhis;
create large tablespace TSDCURRENT pagesize 8 K managed by database
using (
FILE  '/db2data/TSDCURRENTC001.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC002.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC003.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC004.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC005.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC006.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC007.DAT' 16 G,
FILE  '/db2data/TSDCURRENTC008.DAT' 16 G
) autoresize yes extentsize 64 prefetchsize automatic bufferpool tsdbcur;

create large tablespace TSIHISTORY pagesize 8 K managed by database
using (
FILE  '/db2index/TSIHISTORYC001.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC002.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC003.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC004.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC005.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC006.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC007.DAT' 16 G,
FILE  '/db2index/TSIHISTORYC008.DAT' 16 G
) autoresize yes extentsize 64 prefetchsize automatic bufferpool tsibhis;

create large tablespace TSICURRENT pagesize 8 K managed by database
using (
FILE  '/db2index/TSICURRENTC001.DAT' 16 G,
FILE  '/db2index/TSICURRENTC002.DAT' 16 G,
FILE  '/db2index/TSICURRENTC003.DAT' 16 G,
FILE  '/db2index/TSICURRENTC004.DAT' 16 G
) autoresize yes extentsize 64 prefetchsize AUTOMATIC bufferpool tsibcur;
```

3.4 内存的设置管理

从 DB2 V9.5 开始，DB2 采用了多线程模型，相应的内存模型也做了很大改变。新的内存模型使得更多的内存可以作为共享内存使用，并且可以动态调整，这为数据库的内存配置提供

了极大的便利。这一节关于内存模型的介绍主要是基于 DB2 V9.5 及以上版本。

3.4.1 解密 DB2 内存

从图 3-10 可以看到 DB2 的各个内存区域，例如 INSTANCE_MEMORY 代表了整个实例可以使用的最大内存数，包括实例共享内存、数据库共享内存、应用程序全局内存和代理私有内存。如果把 INSTANCE_MEMORY 设成 AUTOMATIC，DB2 会根据操作系统的物理内存大小给数据库分配特定大小的内存；DATABASE_MEMORY 代表了数据库共享内存的大小；APPL_MEMORY 代表了应用程序全局内存的大小。

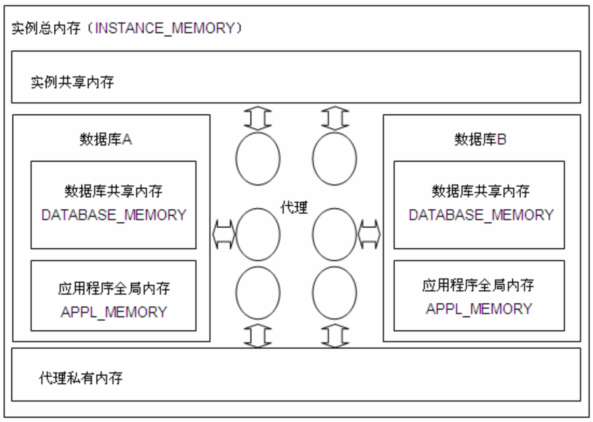


图 3-10 DB2 内存模型

在了解了 DB2 内存的整体结构以后，接下来分别谈谈这些内存区域的具体含义。

1. 实例共享内存

实例共享内存是指实例级的通用任务所需要的内存，比如数据库监控、审计、节点（分区）间通信等。实例共享内存在实例启动时分配，在实例停止时释放。实例共享内存的结构如图 3-11 所示：

图 3-11 DB2 实例共享内存图

数据库监控堆主要用于一些监控活动，是对数据库捕获快照信息的基础；审计缓存用于有关数据库审计的操作；快速通信管理器（FCM）用于不同数据库节点（分区）之间的通信。

2. 数据库共享内存

数据库共享内存是指数据库一级任务所需要的内存单元，这些内存单元被数据库进程共享，用来满足客户请求。这些内存在数据库被激活时分配，在数据库被关闭时释放。数据库共享内存的结构如图 3-12 所示。

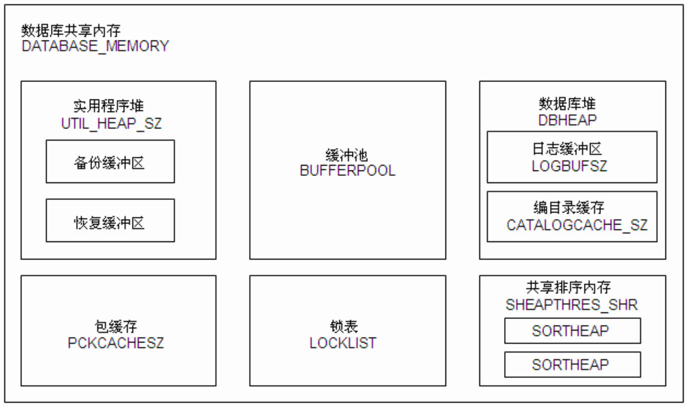


图 3-12　DB2 数据库共享内存图

从图 3-12 可以看到数据库共享内存里面有若干区域，用于满足数据库操作的各种需求，这些内存区域的大小基本上都是在数据库配置参数（DB CFG）中设置的，下面分别进行解释。

（1）实用程序堆（UTIL_HEAP_SZ）

为实用程序提供了内存，比如备份和恢复用的缓冲区都是从实用程序堆里面分配的，还有 LOAD 等实用程序也会用到实用程序堆。

（2）缓冲池（BUFFER POOL）

用来缓存数据或索引的内存区域，是磁盘数据在内存中的映射，相应页大小的缓冲池和相应页大小的表空间对应，如果没有设置相应页大小的缓冲池，DB2 为了能让数据库启动不受影响，会自动创建几个不同页大小的默认缓冲池。

数据从磁盘到缓冲池是由 DB2 的预取进程（db2pfchr）完成的。如果 db2agent 预见到 SQL 语句需要读取大量的连续数据，可以在真正需要数据之前预先从磁盘读取数据。

对数据的修改都会在缓冲池里进行，而页面清除进程（db2pclnr）负责把修改过的数据页回写到磁盘。清除进程一般是在特定条件下触发的，比如缓冲池里的修改页超过一定百分比（由 CHNGPGS_THRESH 参数决定）或者数据落实磁盘落后于日志落实磁盘超出一定量时（由 SOFTMAX 参数决定）。

（3）数据库堆（DBHEAP）

主要包括日志缓冲区和编目录缓存。日志缓冲区用于日志读写；编目录缓存包含最近用到的数据库编目录信息，包括表的定义、统计信息和权限等。

（4）包缓存（PCKCACHESZ）

保存了最近使用的 SQL 语句或者存储过程的查询计划，如果有相同的查询，可以直接从包缓存中获得查询计划，而不需要重新编译。

（5）锁表（LOCKLIST）

保存了当前所有锁的信息。

（6）共享排序内存（SHEAPTHRES_SHR）

是数据库进行排序的场所，比如连接等操作会用到排序内存。如果排序内存不够用，排序会放到磁盘上进行，这种情况被称作排序溢出，它会影响排序的效率。

在 DB2 中与共享排序内存相关的参数有两个，一个参数是 SORTHEAP，它规定了每一个排序可以使用的最大内存量。另一个是 SHEAPTHRES_SHR，它规定了所有共享排序可以使用的总内存的软限制，当共享排序内存接近这个值，数据库将会减少共享排序内存的分配，当共享排序内存达到或超过这个值，则数据库只给排序分配最少的内存让其能够完成手头上的工作，而新申请的排序则会报 SQL0955C 错误。因此调整 SORTHEAP 的同时一定不要忘了调整 SHEAPTHRES_SHR 的大小。

上面提到的内存区域中，缓冲池、包缓存、锁表以及排序内存等可以在数据库共享内存的限制下，通过自调优内存管理（STMM）互相协调，自动调节大小，以满足数据库的需要。关于 STMM，将在 3.4.4 节具体讲述。

3. 应用程序全局内存

这里的应用程序全局内存并不是外部应用程序自己的内存，而是外部应用程序连接到数据库以后，数据库为了处理这些应用程序的数据请求而分配的内存，总量不能超过 APPL_MEMORY 的限制。当然，也可以把 APPL_MEMORY 设制为默认值 AUTOMATIC，这样数据库可以自动调整。应用程序全局内存的结构如图 3-13 所示。

图 3-13　DB2 应用程序全局内存

下面对其进行具体介绍。

- 应用程序堆（applheapsz）：规定了所有应用程序可以消耗的程序堆的内存大小。
- 统计信息堆（stat_heap_sz）：为运行 RUNSTATS 命令提供了内存区域，用来收集数据库统计信息。
- 语句堆（stmtheap）：为 SQL 语句或者 XQUERY 查询语句在编译时提供了内存区域。

4. 代理私有内存

每个 DB2 代理进程都需要一定的内存完成其工作，比如构建查询计划、排序、记录游标以及收集统计信息等。在 DB2 V9.1 和之前版本中，代理数量的上限由 maxappls 参数和 maxagents

参数的较小者决定。从 DB2 V9.5 开始，maxappls 默认为 AUTOMATIC 并且 maxagents 不再被使用，DB2 可以根据系统工作负载和资源情况来决定可以接受的最大应用程序连接数，这大大简化了代理私有内存的管理。DB2 中代理私有内存的结构如图 3-14 所示。

图 3-14　DB2 代理内存图

该结构中各种内存区域的具体含义如下。

（1）代理栈（AGENT_STACK_SZ）

一般 SQL 语句越复杂，代理栈的内存需求量就越大。

（2）私有排序内存（Sheapthres）

sheapthres 是个实例级参数，表示整个实例范围内私有排序总内存的软限制，如果私有排序内存使用超过这个限制，则新请求的私有排序内存分配会大幅减少。如果 sheapthres 被设置为 0 值，则 DB2 使用共享排序。如果 sheapthres 被设置为非 0 值，表示 DB2 将使用私有排序，排序内存在代理中分配，SORTHEAP 参数此时则规定了每个私有排序最多使用的内存量，IBM 信息中心的推荐是将 sheapthres 设置为 SORTHEAP 的整数倍，至少为实例中最大 SORTHEAP 值的 2 倍。

SHEAPTHRES_SHR 与 sheapthres 很容易混淆，为了便于大家记忆，下面列出了它们两者之间的主要区别：

- SHEAPTHRES_SHR 中的 SHR 代表 Share，表示共享内存的限制，而 sheapthres 代表私有内存的限制。
- SHEAPTHRES_SHR 是数据库级参数，代表数据库范围的共享排序内存限制，而 sheapthres 是实例级参数，代表实例范围的私有排序内存限制。
- SHEAPTHRES_SHR 与 sheapthres 是冤家，如果 sheapthres 为 0，DB2 使用共享排序，SHEAPTHRES_SHR 才起作用，如果 sheapthres 不为 0，SHEAPTHRES_SHR 基本就没用了，除非开启了内部并行度（intra_parallel 为 YES）或者 max_connections 参数的值大于 max_coordagent，这时会使用共享排序。
- SHEAPTHRES_SHR 与 sheapthres 都是可以在线调整的，但 sheapthres 在从 0 调到非 0 值时需要重新启动实例才能生效。

（3）JAVA 堆（JAVA_HEAP_SZ）

与 JAVA 相关的内存。

（4）客户端 I/O 块（RQRIOBLK）

与客户端之间进行通信的内存区域。

如果要查看当前内存的分配状况，可以使用 db2mtrk 工具，比如，如果想看实例以及数据库的内存分配状况，可以使用“-i -d –v”选项，下面是一个具体的例子。

```
[db2inst1@db2 ~]$db2mtrk -i -d -v
Tracking Memory on: 2011/06/02 at 22:40:47

Memory for INSTance

   Other Memory is of size 15466496 bytes
   FCMBP Heap is of size 851968 bytes
   Database Monitor Heap is of size 327680 bytes
   Total: 16646144 bytes

Memory for database: LDE
   Backup/Restore/Util Heap is of size 65536 bytes
   Package Cache is of size 20840448 bytes
   Other Memory is of size 196608 bytes
   Catalog Cache Heap is of size 5439488 bytes
   Buffer Pool Heap （1） is of size 969867264 bytes
   Buffer Pool Heap （System 32k buffer pool） is of size 851968 bytes
   Buffer Pool Heap （System 16k buffer pool） is of size 589824 bytes
   Buffer Pool Heap （System 8k buffer pool） is of size 458752 bytes
   Buffer Pool Heap （System 4k buffer pool） is of size 393216 bytes
   Shared Sort Heap is of size 1966080 bytes
   Lock Manager Heap is of size 33423360 bytes
   Database Heap is of size 73990144 bytes
   Application Heap （7648） is of size 65536 bytes
   Application Heap （55） is of size 393216 bytes
   Application Heap （54） is of size 131072 bytes
   Application Heap （53） is of size 65536 bytes
   Application Heap （52） is of size 65536 bytes
   Application Heap （50） is of size 65536 bytes
   Application Heap （51） is of size 65536 bytes
   Application Heap （49） is of size 65536 bytes
   Application Heap （48） is of size 65536 bytes
   Application Heap （47） is of size 327680 bytes
   Application Heap （46） is of size 65536 bytes
   Application Heap （45） is of size 65536 bytes
   Application Heap （44） is of size 65536 bytes
   ……………………………………… 省略
   Applications Shared Heap is of size 1245184 bytes
   Total: 1113653248 bytes
```

3.4.2　缓冲池设计

图 3-15 所示为 DB2 缓冲池的工作机制，主要包括两种类型的缓冲池，基于页的缓冲池和基于块的缓冲池。

第一种，基于页的缓冲池，这是默认方式。缓冲池为数据库页提供工作内存和高速缓存，当应用程序访问表行时，数据库管理器将在缓冲池中查找包含该行的页。如果在缓冲池中找不到该页，那么数据库管理器将从磁盘中读取该页并将其放入缓冲池。然后，可以使用该数据来处理查询。因此，数据库管理器需要从磁盘读取或写入磁盘的次数越少，性能就越好。

第二种，基于块的缓冲池。基于块的缓冲池可以将块读入相邻的内存区而不是将它分散装

入单独的页，以此来提高预取操作的效率。每个缓冲池的块大小必须相同，并且由 BLOCKSIZE 参数进行控制。

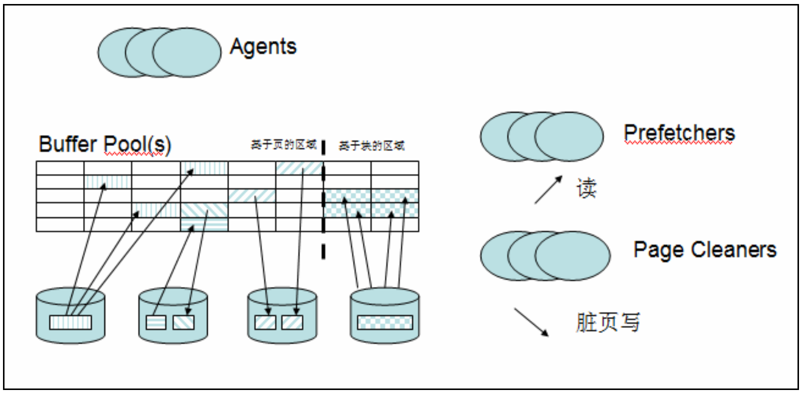


图 3-15　DB2 缓冲池工作机制

> **注意：基于块的缓冲池适用场景**
>
> 对于连续读写频繁的 OLAP 查询可以得益于基于块的缓冲池。默认情况下，所有缓冲池都是基于页的，这意味着预取操作将把磁盘上相邻的页放入到不相邻的内存中。而如果采用基于块的缓冲池，则 DB2 将使用块 I/O 一次将多个页读入缓冲池中，这样可以显著提高顺序预取的性能。

可以使用 CREATE BUFFERPOOL、DROP BUFFERPOOL 和 ALTER BUFFERPOOL 语句来创建、删除和修改缓冲池。SYSCAT.BUFFERPOOLS 目录视图记录定义的缓冲池信息。下面介绍 DB2 缓冲池设计中需要重点关注的三个方面。

1. 为缓冲池分配内存

对于 64 位系统而言，缓冲池的大小没有最大值的限制，仅受限于物理内存的大小。缓冲池越大越好，但是，在某一点，当命中率接近 100%的时候，再增加内存，对提高命中率收效甚微。

对于 OLTP 类型的系统，一开始将缓冲池的初始值设为 50%的可用内存，然后再根据系统实际运行情况进行调整。

对于 OLAP/DSS 类型的系统，可以将 60%左右的可用内存分配给缓冲池。

2. 调整缓冲池大小

（1）创建数据库时，DB2 会自动创建一个名为 IBMDEFAULTBP 的默认缓冲池，所有的表空间都共享该缓冲池。

（2）仅当 SYSCAT.BUFFERPOOLS 中 NPAGES 值为－1 时，数据库参数 BUFFPAGE 才会决定缓冲池的大小；否则，BUFFPAGE 参数不会起作用，DB2 会用 NPAGES 参数所指定的页数创建缓冲池。

（3）当 SELF_TUNING_MEM 参数打开时，使用 AUTOMATIC 选项的缓冲池将自动调节大小。

（4）除了 DB2 默认的缓冲池 IBMDEFAULTBP 外，还可以创建自定义缓冲池。

下面的 CREATE BUFFERPOOL 语句创建页大小为 4KB 的缓冲池 BP1：

```
CREATE BUFFERPOOL "BP1" IMMEDIATE  SIZE 1000 AUTOMATIC PAGESIZE 4 K
```

可以用 ALTER BUFFERPOOL 语句将缓冲池 BP1 从 1000 个页面增加到 2500 个页面，其中 IMMEDIATE 表示立即生效。

```
ALTER BUFFERPOOL BP1 IMMEDIATE SIZE 2500
```

3. 配置多缓冲池

对配置多缓冲池有如下建议：

（1）数据表空间和索引表空间使用不同的缓冲池，方便管理和提高性能。

（2）针对不同应用特点量身定做缓冲池。

对于 OLTP 应用，可以根据业务特点使用多个缓冲池，以便长时间缓存应用所使用的数据页；对于 OLAP 应用，建议采用大的缓冲池，以利于大块数据顺序读取。

（3）为频繁使用的热表设立单独的缓冲池。

通常一个数据库系统中，都会有许多频繁使用的热表，如果这些表的数据全都位于缓冲池中，那么访问起来就非常快。现在假设有一个大表和这些热表共享一个缓冲池，那么 DB2 引擎处理针对这个大表的查询时，很可能将这些热表的数据页从缓冲池中交换出去，如果再次需要这些热表数据时就必须重新从磁盘读取。所以，建议用单独的缓冲池来存储这些热表数据。

3.4.3　缓冲池命中率

一个表空间只能与一个缓冲池相关联，而一个缓冲池则可以用于多个表空间。缓冲池命中率用来描述数据库管理器不需要从磁盘装入页（即该页已经在缓冲池中）就能处理页请求的百分比。下面是缓冲池命中率的公式：

```
(1 - ((数据物理读次数+ 索引物理读次数) /(数据逻辑读次数 + 索引逻辑读次数))) * 100%
```

接下来用例子说明一下上面公式的用法，下面的内容是缓冲池 IBMDEFAULTBP 的快照信息，从中可以看到数据物理读次数为 86，索引物理读次数为 62，数据逻辑读次数为 194，索引逻辑读次数为 106，根据上面的公式，可以计算出缓冲池 IBMDEFAULTBP 的命中率为 51%。

```
Bufferpool name                    = IBMDEFAULTBP
Database name                      = DBONE
Database path                      = /home/test/test/NODE0000/SQL00001/
Input database alias               = DBONE
Snapshot timestamp                 = 08/16/2010 14:16:48.753322
Buffer pool data logical reads             = 194
Buffer pool data physical reads            = 86
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool data writes                    = 0
Buffer pool index logical reads            = 106
Buffer pool index physical reads           = 62
Buffer pool temporary index logical reads  = 0
```

从实践上来看，缓冲池的命中率应当超过 95%，并尽可能接近 100%，这样使用磁盘 I/O 的频率就越低。要提高缓冲池命中率，可以参考下面这些方法：

（1）为缓冲池分配足够内存。

（2）为每个表空间分配一个独立的缓冲池，这样容易管理，也有助于性能提升。

（3）根据表的访问热度分配缓冲池，即冷表和热表使用不同的缓冲池。这样可以减少缓冲池和磁盘之间数据的频繁交换。

3.4.4 STMM 自动管理内存

有过 DB2 内存调优经验的用户都知道，如果依靠人力进行内存的实时调整，通常效果都不会太好。

从 DB2 V9 开始引入了一种新的内存管理功能，叫做自调优内存管理（STMM），STMM 通过监测多个内存区域的使用情况来自动调整其大小，它特别适合在数据库负载经常变化或者系统资源状况比较复杂的场合使用，从而大大简化数据库管理员的工作。

例如，如果排序操作需要更多的内存，而一些缓冲池又有多余的内存，那么 STMM 会释放多余的缓冲池内存，并将它分配给排序堆。STMM 可以管理 DB2 实例（INSTANCE_MEMORY）和数据库（DATABASE_MEMORY）级别的内存，包括：缓冲池、包缓存、锁内存区、排序内存区和数据库共享内存区等。

STMM 对内存的调整过程都被写入数据库分析日志中（db2diag.log）和 STMM 日志（stmm.log）。

DB2 默认在单分区环境下启用 STMM，在多分区环境下关闭 STMM。如果不需要在单分区环境下启用 STMM，则可在创建数据库之后将数据库配置参数 self_tuning_mem 设置为 OFF：

```
update db cfg for database <数据库名> using SELF_TUNING_MEM OFF
```

当启用 STMM 时，至少需要将两个或两个以上的内存参数设置为自动（Automatic），否则尽管 SELF_TUNNING_MEM 设置为 ON，STMM 也是没有办法动态调整内存的。

> **小问题：STMM 能调整不同页大小的缓冲池内存吗？**
>
> 答案是可以，因为 DB2 会自动实现不同页面大小的内存转化。

总之，STMM 可以实时检测并优化 DB2 的内存使用情况，包括应用、实例、数据库等，这样大大减少了 DBA 的工作量。

3.4.5 为电信公司设计缓冲池

在实践中，DBA 需要根据实际情况选择合适的缓冲池大小。此外，对于数据库中许多频繁使用的热表，可以考虑将这些热表用单独的表空间存储，并设置单独的缓冲池。

针对这个电信公司的应用负载特点，我们做了如下的缓冲池规划：

（1）开启 STMM，允许 DB2 可以自动调整内存大小。

（2）定义缓冲池的页大小为 8KB，与表空间页面大小相匹配。

（3）针对数据和索引创建不同的缓冲池，避免彼此之间的影响，从而提高性能。

（4）针对数据的冷热程度，定义不同的缓冲池。

（5）初始时使用 50%的可用物理内存用作缓冲池，随后根据系统运行情况再做调整。数据库服务器的可用内存为 64GB，预先分配 32GB 内存供缓冲池使用。

最终，规划了 4 个缓冲池，分别为 tsdbhis、tsdbcur、tsibhis 和 tsibcur，它们的页大小都是 8KB，并使用 STMM 内存自调优，如表 3-6 所示。

小问题：为什么不同缓冲池的初始化大小不一样？

读者可以发现，tsdbcur 和 tsibcur 的初始化大小都比较大，分别为 16GB 和 8GB，主要是考虑到当前数据使用频繁，所以缓冲池越大越好。tsdbhis 和 tsibhis 的初始化大小就较小了，这是由于历史数据平时用的不多，万一真的有大表查询需要大的缓冲池，也可由 STMM 来动态调整。

表 3-6　缓冲池规划

缓冲池	初始大小	页大小	STMM 是否激活	用途
tsdbhis	4GB	8KB	是	历史冷数据
tsdbcur	16GB	8KB	是	当前热数据
tsibhis	4GB	8KB	是	历史冷数据索引
tsibcur	8GB	8KB	是	当前热数据索引

创建缓冲池的语句如下：

```
create bufferpool TSDBHIS immediate size  512000  pagesize 8 K;
create bufferpool TSDBCUR immediate size  2048000  pagesize 8 K;
create bufferpool TSIBHIS immediate size  512000  pagesize 8 K;
create bufferpool TSIBCUR immediate size  1024000  pagesize 8 K;
```

3.5　高质量物理设计最佳实践

在写这本书之前，我们和一些客户方的资深 DBA 交流过，大家认为所谓物理设计，就是要通过调整让 DB2 把硬件的处理能力发挥出来，这其实说出了本质。按照这个思路，本章讲述了让 DB2 发挥硬件处理能力的方方面面，包括硬件选型、存储设计、表空间设计和内存设置等。为了进一步方便读者，我们接下来将分享物理设计的最佳实践。

3.5.1　硬件配置

关于硬件的配置，请参看本章 3.1 节，这里不再赘述，仅做总结如下：

以 Power 系列主机为例，1 个 Power 处理器内核对应 8GB 内存的配置方案可以满足绝大多数的应用；采用 RAID 技术的存储设备中，1 个 Power 处理器内核可以处理 300~500GB 活动数据。

3.5.2　存储的设计配置

对于一个数据库来说，需要规划这几个重要的存储空间，包括，实例目录、数据库目录、

数据、活动日志、归档日志、备份以及临时数据等。这里仅分享数年来总结的最佳实践。

（1）存储的性能取决于物理磁盘的个数，优先考虑磁盘的个数而不是容量。

（2）在磁盘个数有限的情况下，建议将数据和索引都跨越所有物理盘，这样可以发挥多块盘的 I/O 并行处理能力。但从维护角度讲，在建表时仍建议为数据、索引和大对象创建独立的表空间。

（3）最多两次条带化，存储层通过 RAID 实现，操作系统层通过 LVM 实现，DB2 通过表空间容器实现，建议在存储层和数据库层走条带。

（4）选择正确的 RAID 级别，对于 OLTP 事务型应用，以随机小块读写为主，可以考虑选择 RAID 10；对 OLAP 分析型应用，以连续大块读写为主，可以考虑选择 RAID 5。

（5）数据和日志应放置于不同的物理盘上。对于数据，可以选择 RAID 5 或者 RAID 10；对于活动日志，由于写操作很频繁，而且对安全性要求高，可考虑选择 RAID 10。

（6）存储的驱动程序尽可能支持多路径 I/O，它支持 I/O 子系统的负载均衡和流量控制。

3.5.3 数据库参数初始化选择

数据库参数的配置是非常繁琐的工作，如果存在相似的运行良好的系统，以此系统的配置参数为基础，并进行逐步调整，可以达到事半功倍的效果。在没有类似系统的情况下，也可以使用 DB2 的配置向导（**AUTOCONFIGURE 命令**）对系统进行初步配置后，再根据数据库系统的自身特性进行针对性调整，以达到最优的参数配置。

在参数调整的过程中，需要额外注意下面这些性能相关的参数：

- 参数 newlogpath 决定事务日志的存放位置，将事务日志放置在专用的存储设备上，避免和数据 IO 竞争。
- 参数 logbufsz 决定事务日志的缓冲大小。通常默认值太小，根据实际情况，将值设置在 1000 个页以上可以获得较好性能。
- 参数 mincommit 可以让 DB2 实现批量处理多个提交的事务。对于特别繁忙的 OLTP 系统，可以将其值调整为大于 1 ，例如 3，这样可以实现批量提交。
- 参数 diagpath 用于指定 DB2 诊断日志的位置，强烈建议在 DPF 分区数据库中，使用分区的本地路径，而不是 NFS 路径。

关于 DB2 其他参数的调整，请参考本书第 7 章 7.1 节的相关内容。

3.5.4 内存设置管理

DB2 的内存自动调整机制（STMM），可以动态调整数据库内存，对于多数系统而言，使用 STMM 能够取得很好的性能。然而，对于 DPF 分区环境而言，使用 STMM 要格外小心。

在 DPF 分区环境中，如果各分区中数据量分布、内存大小以及分区的负载都基本一致，STMM 将会非常有效；否则就需要在这些分区上禁用 STMM。例如 STMM 应该在编目分区上禁用。

总的来说，不建议在 DPF 分区数据库中启用 STMM。

3.5.5 表空间设计管理

表空间的设计与存储设计息息相关，在设计存储时，应该为后来的表空间设计做好铺垫。

总的来说，表空间的设计应该遵循以下原则：

- 对于临时表空间，建议使用 SMS；其他数据建议使用 DMS，以发挥性能优势。
- 对 DMS 表空的容器类型，建议使用文件系统，而不是裸设备。
- 为大表和小表单独分配独立的表空间；表和索引分配不同的表空间。
- 热度不同的表或索引使用不同的表空间。
- 当创建 DMS 表空间时，如果存储的是普通数据，可指定 NO FILE SYSTEM CACHING 选项，如果是大对象 LOB 数据，可考虑利用文件系统缓存。

3.6　小结

物理设计是数据库系统的基石，只有在优秀的物理设计基础上才可能搭建出一个高性能的数据库系统。本章以主机、存储等方面为主线，从硬件选型、存储设计、表空间设计、内存的管理与使用等层面逐个进行深入地讲解和分析，最终总结了物理设计的最佳实践。通过本章学习，希望读者能够全面、深刻地理解数据库的物理设计，并能将设计案例活学活用到日常工作中去。

第4章 经典逻辑设计

作为一名 IBM 原厂的数据库顾问，笔者经常参加客户、IBM 和开发商的工作会议，笔者发现大部分与会人员也包括笔者自己常常在会议中口若悬河。但是，笔者记忆犹新的是在西安参加的一次会议上，有一家来自兰州的开发商，他们的人员包括公司老总在内，却保持着相当的沉默，透露着一丝紧张和青涩。这家开发商并不大，员工也朴实无华，正因为这样，他们设计的系统，开发的代码也体现了这种返璞归真的风格：对 DB2 的经典逻辑设计包括表设计和索引设计运用很见功底，编写的代码也没有大公司开发的那么绚丽多彩，但是扎实可靠地反映了业务需求和性能需求。这样的系统才是一个契合自上而下方法学的系统，就像一只具有极大升值空间的潜力股，我相信如果将来运用更高级的设计技术，运行在更好的硬件平台上，在运维阶段一定能表现出卓越不凡的性能。

不论对 Oracle 还是 DB2，其经典逻辑设计的精髓就是灵活运用规范化理论设计表，并通过各种表连接技术和索引技术加速数据的查询与处理。本章将重点阐述 DB2 的经典逻辑设计，先从业务需求和性能需求谈起，然后解答现场中遇到的较多的问题：如何设计常规表、临时表和时态表？如何设计索引？如何设计触发器？

4.1　数据库设计不仅是一种艺术

数据库的前期设计至关重要，它决定了数据库的基因，也决定了数据库后期运维成本。我作为“救火”队员经常去客户现场，见过很多数据库，一些数据库的设计不合理，导致在运维阶段出现问题时分析和解决的代价非常高昂。

小故事：扁鹊三兄弟谁的医术更高明？

魏文王问名医扁鹊：“你家兄弟三人，都精于医术，谁的医术最好呢？”
扁鹊答：“大哥最佳，二哥次之，我最差。”
文王再问：“那为什么你最出名呢？”
扁鹊答：“大哥治病于病情发作之前，诊疗前后无甚感觉，一般人不知他事先已除病因，所以他名气全无；二哥治病于病情初起之时，一般人以为他只能治小病，所以他的名气只及本乡；我是治病于病情严重之时，人们都看到我在经脉上下针放血、用药教药，都以为我医术高明，因此名气响遍全国。”
我一直很喜欢这段话，也一直在思考扁鹊的话。起死回生、妙手回春固然高明，高瞻远瞩、未雨绸缪才是真正的高手。类似的，发生火灾时，消防队员的工作当然重要。可是要是监察队工作到位就不会有火灾了。反过来，如果建筑物的设计本身存在缺陷，出现火灾时，消防队员也只能望洋兴叹。

4.1.1　业务需求

业务需求是逻辑设计需要满足的首要目标。这也许是老生常谈了，但实际上，很多设计人员在项目紧张时仅重视应用的功能，却忽视了表和索引的设计质量：很多时候在设计表的时候，竟然没有主键，没有索引，也没有为它指定表空间。出现这种情况的原因在于，缺乏数据库设计经验，另外通常项目都很紧，要赶工期，所以表和索引的设计质量就没有放在重要位置上了。等到系统上线后慢慢出现各种问题，这时再想来调整表的结构就不是那么容易的了。

在数据建模时，必须明确业务需求，确保数据忠实地、一致地反应业务活动，这关系到数据库模型的质量，所以良好的模型和数据库设计是高质量应用系统的重要基础。

我曾经遇到一个用户抱怨说：“DB2 太弱了，在 DB2 中建一个含有 300 多个 VARCHAR（40）字段的表都不行。”我当时听了后非常震惊，到底是怎样特殊的应用才需要有这样特殊的表？我相信有这样特殊的应用，但是否真正考虑过下面的建议：

- 对应用的业务需求是否有深刻的了解，对业务需求的抽象程度是否恰当？
- 300 多个 VARCHAR（40）字段的数据长度是否过于草率？
- 对这种特殊的表是否进行了正确的规范化？

4.1.2　性能需求

如今，很多企业需要处理的数据量都在爆炸式增长，信息系统仅仅做到功能正确是远远不够的，还必须足够快。作为信息系统基石的数据库，它的性能至关重要。但是，大多数人仅关

注业务需求，数据库性能不是他们关注的重点，等系统上线后性能调优应该交给资深 DBA 去摆弄，他们调整几个“有魔力”的数据库参数，于是性能就提高了。如果还是不够快，就该升级硬件了。

设计时总想着“先把系统搞出来，之后再让专家在生产环境下调优”是错误的，性能专家不是魔法师，性能调优也不是万能的，因为调优往往不能修改应用程序和系统结构，而且有时候性能调优并不能从根本上解决性能问题，甚至会带来新的问题。

性能调优真正能做的主要包括两个方面：一方面，调优系统整体性能，主要是根据当前 CPU 能力、可用内存、IO 子系统的资源情况来设置相应参数。另一方面，针对具体查询的调优，比如增加或纠正索引方案，为复杂查询创建 MQT 等。

总之，性能问题的罪魁祸首，通常是不合理的设计。这就要求在设计数据库时就开始考虑性能，真正按照自上而下方法学的要求指导设计工作，后续的调优只是一种必要的补充手段。

4.2　常规表设计体现真功夫

应用的数据模型设计以及数据的增删改查，都围绕表进行，表设计是数据库设计中最重要的部分，也最能体现真功夫。

4.2.1　规范化决定成败

优良的数据库，一定是恰当的数据，在恰当的时刻被分到恰当的表中。

规范化为何如此重要？答案在于：规范化使混乱变得有序。成功的数据建模应严格遵守基本的设计原则。当我在大学里学习数据库课程的时候，仅仅为了应付考试，死记硬背规范化这种枯燥、沉闷的东西。多年以后，有了一些数据库方面的使用经验，我才认识到规范化的重要性，才认识到 Codd 的厉害之处。

E. F. Codd 为表的设计制定了游戏规则——范式（Normal Forms），这为数据库逻辑设计指明了方向。规范化的表，既方便对数据的存取操作，也会显著简化应用程序的事务处理、查询、报表等。

规范化的目的在于：

- 消除冗余数据，例如避免在多个表中存储相同的数据。
- 强制数据之间的依赖关系，并把数据拆分到多个相关表中。
- 为数据增长提供最大灵活性。

在数据库设计中，第三范式（3NF）是最重要、运用最广泛的设计范式，第三范式规定：

- 原子性：表的每个字段不可拆分。
- 唯一键：表内的每一行都应该被唯一的标识，即有主键。
- 独立性：表内的非键字段相互独立，非键字段仅依赖于主键。

非规范化的设计可能带来数据冗余，甚至导致数据的不一致性，进而导致复杂的数据输入控制代码、拙劣的性能、模型无法演进等问题。不过，有时候为了提高性能，在逻辑设计中也可以考虑反规范化（de-normalize）。反规范化的意思是，打破该表之前遵从的一种或多种范式，利用空间换时间来提高性能。

反规范化最常见的技术是重新连接表来避免过多的表连接操作。通常，规范化过程要求将相关的信息放入不同的表中，应用查询时需要通过多表连接得到结果，但是多表连接操作需要消耗系统资源和时间。所以，反规范化的建议是不用严格按照第三范式将数据拆分到不同表中，而是允许数据有冗余，这样可以减少表连接的个数，从而提高性能。

下面的例子展示了增加冗余字段的反规范化技术，REQ_SUM 表示操作的金额，PROD_PR 为所买产品的价格，PROD_CNT 为所买产品的数量。REQ_SUM 是冗余字段，可以由 PROD_PR 乘以 PROD_CNT 得到，这表明该表的设计不满足第三范式。但是，增加 REQ_SUM 这个冗余字段，可以提高查询速度，这就是以空间换时间的做法。

```
Column                         Type      Type
name                           schema    name               Length  Scale Nulls
------------------------------ --------- ------------------ -------- ----- ------
REQ_ID                        SYSIBM    INTEGER                  4     0 No
USER_ID                       SYSIBM    VARCHAR                 48     0 No
PROD_ID                       SYSIBM    VARCHAR                 48     0 No
PROD_PR                       SYSIBM    VARCHAR                 48     0 No
PROD_CNT                      SYSIBM    VARCHAR                 48     0 No
REQ_SUM                       SYSIBM    VARCHAR                 48     0 No
...
```

比较：规范化和反规范化

关于规范化和反规范化，不能一概而论，需要权衡利弊，灵活使用。不能为了刻意遵循规范化，束手束脚，使系统性能不合理地下降，该采用反规范化的地方就大胆采用。

4.2.2　用户与模式

数据库中的每一个对象（包括表、索引和视图等）都有唯一的名字，名字由模式名（schema）加上对象名构成，即模式名.对象名。这就如一个人的姓名，模式名相当于“姓”，而对象名相当于“名”。重名的情况下，比如“李小明”与“刘小明”，可以通过“姓”将这两个人区分开。

模式有两个作用：

（1）对数据库对象进行逻辑分组，即用模式对不用应用的表进行分组。比如在一个综合性的数据库中，可以将财务相关的表放在一个模式下，人力资源相关的表放在另一个模式下。

（2）在多用户的情况下，避免不同用户使用相同对象名产生的冲突。比如，A 用户创建了表 T1，B 用户也想用 T1 作为表名，有了模式后，这个问题就解决了。

模式在运维和数据迁移中也有各种方便之处。例如，db2look 和 db2move 等工具可以限定只操作某模式下的所有对象。

比较：Oracle 和 DB2 模式的区别

Oracle 的模式名就是用户名，但是 DB2 中的模式名不一定是用户名，DB2 的模式可以是用户名，也可以是用户新创建的模式。另外，Oracle 可以创建数据库用户，DB2 只能使用操作系统用户。

DB2 支持两种方式来创建模式：

（1）显示创建：通过 CREATE SCHEMA 命令创建，例如下面创建模式 tpch。

```
db2inst1@db2ring:->db2 "create schema tpch"
DB20000I  The SQL command completed successfully.
```

（2）隐式创建：在创建对象时指定模式，如果以前没有创建，那么在创建对象时 DB2 会自动创建（当然需要有相应的权限），例如下面创建表 t1 时，指定模式 bcp，由于 bcp 以前没有创建过，所以会自动创建。

```
db2inst1@db2ring:->db2 "create table bcp.t1(id int not null, c1 varchar(10))"
DB20000I  The SQL command completed successfully.
```

那么如何使用模式呢？有以下几种场景：

（1）用户连接数据库后，如果 SQL 语句引用表时没有指定模式，那么默认模式就是用户的 ID，如下面对表 T1 的查询就没有提供模式名，所以使用默认模式。

```
db2inst1@db2ring:/home/db2inst1>db2 "select * from t1"

ID          DAT
----------- ----
          1 abc
          2 cbd

  2 record(s) selected.
```

（2）可以显式用“模式名.表名”来引用表，如下例中的 tpcds.store_sales。

```
db2inst1@db2ring:->db2 "select count(*) from tpcds.store_sales"

1
-----------
   10000000

  1 record(s) selected.
```

（3）可以用 SET CURRENT SCHEMA *schema_name*，来修改当前会话中的默认模式。

```
db2inst1@db2ring:/home/db2inst1>db2 set current schema tpcds
DB20000I  The SQL command completed successfully.

db2inst1@db2ring:/home/db2inst1>db2 "select count(*) from store_sales"

1
-----------
   10000000

  1 record(s) selected.
```

创建好模式后，也可以用 list tables for schema <*schema_name*>查看某模式下的表和视图等，例如下面查看模式 tpcds 下面的所有表。

```
db2inst1@db2ring:->db2 list tables for schema tpcds

Table/View                      Schema          Type  Creation time
------------------------------- --------------- ----- --------------------------
```

```
CUSTOMER                    TPCDS        T    2012-07-28-09.06.12.515357
CUSTOMER_ADDRESS            TPCDS        T    2012-07-28-09.06.02.438131
CUSTOMER_DEMOGRAPHICS       TPCDS        T    2012-07-28-09.06.04.398424
DATE_DIM                    TPCDS        T    2012-07-28-09.06.04.807513
HOUSEHOLD_DEMOGRAPHICS      TPCDS        T    2012-07-28-09.06.13.200210
INCOME_BAND                 TPCDS        T    2012-07-28-09.06.09.492320
ITEM                        TPCDS        T    2012-07-28-09.06.09.813721
PROMOTION                   TPCDS        T    2012-07-28-09.06.13.765352
STORE                       TPCDS        T    2012-07-28-09.06.11.022605
STORE_SALES                 TPCDS        T    2012-07-28-09.06.14.407324
TIME_DIM                    TPCDS        T    2012-07-28-09.06.08.247587
```

除了用户自定义模式外，还有一些系统模式是供 DB2 内部使用的，主要有以下几个组：

（1）SYSIBM 模式下的对象存储编目信息，即类似 Oracle 中的数据字典。

（2）SYSCAT 模式下的对象是系统编目视图，通过它们可以查询数据库对象的各种信息。

（3）SYSSTAT 是统计视图模式，为 DB2 优化器提供统计信息。

（4）SYSIBMADM 是系统管理视图模式，DB2 V9 引入，用于监控和管理数据库。

4.2.3　字段与类型

在设计表结构时，选择合适的数据类型非常重要，这不仅直接影响存储的空间占用，而且关系到性能。

首先需要搞清楚 DB2 提供的内置类型系统，它是 DB2 的基础，本小节将用与 Oracle 对比的方式，讲述它们之间的区别。

DB2 的内置数据类型分为三类：

- 数值类型
- 字符类型
- 日期类型

如图 4-1 所示，列出了 DB2 的三类内置数据类型，其中字符类型是广义的，从横向上看，它包含单字节字符、双字节字符、二进制数据和 XML；从纵向上看，它包含定长字符类型、变长字符类型、长字符类型和大对象 LOB（Large Object）。接下来，我们具体讲一下这三种类型。

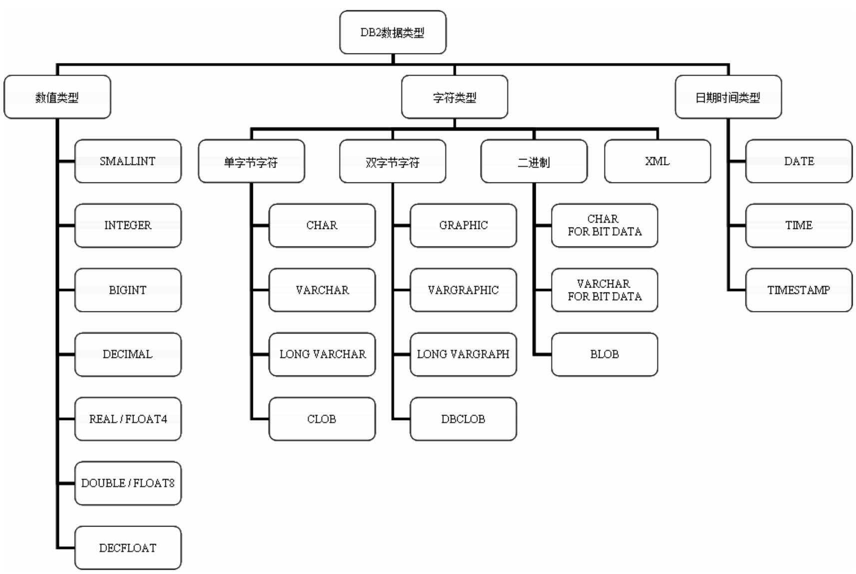


图 4-1　DB2 类型系统

1. 数值类型

DB2 的数值类型是多样化的。表 4-1 列出了 DB2 的数值类型，并给出了对应的 Oracle 数据类型 Number。当然，这种对应只是表现形式上的，两者内部的存储格式大相径庭。

表 4-1　DB2 数值类型与 Oracle 比较

DB2 数据类型	描述	Oracle 数据类型
SMALLINT	小整型，2 字节，十进制精度为 5 位，范围-32,768 到+32,767	Number(5)
INT(或 INTEGER)	整型，4 字节，十进制精度为 10 位，范围 -2,147,483,648 到 2,147,483,647	Number(10)
BIGINT	大整型，8 字节，十进制精度为 19 位	Number(19)
FLOAT(n)	当 n 在 1～24 时为单精度浮点数（4 字节），当 n 在 25～53 时为双精度浮点数（8 字节），默认为双精度	Number
REAL	单精度浮点数，等价于 Float（24）	Number
DOUBLE	双精度浮点数，等价于 Float（53）	Number
DECIMAL(p,s)	十进制类型，p 表示总位数，s 表示小数点后的位数	NUMBER(p,s)
NUMBERIC	DECIMAL 的同义词	NUMBER
DECFLOAT(16/34)	DB2 V9.5 引入的十进制浮点数，IEEE 标准，精度为 16 或者 34，默认为 34	NUMBER

2. 字符类型

如表 4-2 所示，DB2 和 Oracle 的字符类型基本上能对应起来。另外，再提供几点特别建议，供读者参考。

（1）除了二进制大对象，可以将 FOR BIT DATA 子句用在 CHAR、VARCHAR 和 LONG VARCHAR 上，表示这些类型用于存储二进制数据。例如，CHAR（16） FOR BIT DATA 可用于存储 16 个字节的二进制数据。

（2）LONG 数据类型已经不推荐使用了，它的存储长度与 VARCHAR 和 VARGRAHPIC 差不多，但是 LONG 数据不能缓存在缓冲池中，访问时需要从磁盘中直接读取，因此性能较差。

（3）DB2 的 XML 类型使用原生的 pureXML 技术，有着相对较好的性能。

表 4-2　DB2 的字符类型与 Oracle 比较

DB2 类型	描述	Oracle 内置类型
CHAR(n)	定长字符串，n<=254，n 缺省为 1	CHAR(n)
VARCHAR(n)	变长字符串，n<=32,672	VARCHAR(n)或 VARCHAR2(n)
GRAPHIC(n)	定长双字节字符串，n<=127	NCHAR
VARGRAPHIC(n)	变长双字节字符串，n<=16,336	NVARCHR2
BLOB	二进制大对象，可达 2GB	BLOB

(续表)

DB2 类型	描述	Oracle 内置类型
CLOB	字符大对象，可达 2GB	CLOB
DBCLOB	双字节字符大对象	NCLOB
LONG VARCHAR	长字符串对象，不与数据行存在一起，存储为独立的对象，大小为 32700	LONG
LONG VARGRAPHIC	长双字节字符串对象，最多能存储 16350 个字符	LONG
XML	XML 数据	XMLType

注意：Oracle 的 VARCHAR2 类型

VARCHAR2 是 Oracle 自定义的一个非工业标准 VARCHAR，VARCHAR2 可以用 null 代替空字符串。

3. 日期类型

如表 4-3 所示，DB2 提供三种类型：DATE、TIME 和 TIMESTAMP，其中 TIMESTAMP 精确到毫秒。Oracle 提供两种基本的日期类型：DATE 和 TIMESTAMP，DATE 类型包含日期和时间，也就是 DB2 中 DATE 和 TIME 类型的结合体，其默认格式为 2011-04-26 15.21.20。Oracle 的 TIMESTAMP 类型的最高精度可达到纳秒级，如 2011-04-26.15.21.20.000000，默认精度和 DB2 的时间戳类型一样。从 DB2 V9.7 开始，TIMESTAMP 也支持可选的精度，最大可为 12。

除了 4-3 表中所列的日期格式外，根据实际需要，Oracle 和 DB2 都可以使用多种不同的日期格式。

表 4-3　DB2 的日期类型与 Oracle 比较

DB2	描述	例子	对应 Oracle 类型
DATE	年-月-日	2011-04-26	DATE
TIME	时分秒，HH:MM:SS	15:21:20	
TIMESTAMP	年月日时分秒毫秒	2011-04-26.15.21.20.000000	TIMESTAMP

了解了 DB2 的类型系统后，下面分享一些最佳实践：

（1）用直观简短的编码来代替较长的值。例如，可以用 M 和 F 分别表示男和女，可以用 OPN、CLS 和 SLP 分别表示帐号已开启、关闭和休眠的状态，可以用 CN、US 和 JP 分别表示中国、美国和日本。

（2）在 DB2 中，数值类型的处理性能比字符类型要高，因此尽量选择数值类型。当然，这需要在数据库中用相应的属性来描述这些数值的属性信息。

（3）选择整型类型时，需要注意类型的表示范围，比如 small int 的取值上限是 32767，如果要存储超过 32767 的值，就需要使用 INTEGER 类型。如果需要存储的值甚至超过 BIGINT 的表示范围，可以使用 DECIMAL（n, 0），n 为十进制的位数。

（4）浮点类型表示的数据是近似值，如果在银行金融领域需要精确的计算结果，可以使用 DB2 V9.5 引入的 DECFLOAT。

（5）一般根据字符串长度是否定长来选择 CHAR 和 VARCHAR，需要注意 CHAR 的最大长度是 254。对于长度变化比较大的字符类型，一般选择 VARCHAR。

（6）对于 LOB 和 XML，可以有选择地使用内联（inline）方式来处理，这样较小的 LOB 和 XML 将与关系数据存储在一起，并能预取到缓冲池中，可以提高性能。

4.2.4 建表的讲究

先用一个例子来说明，如下所示，一共创建了 ITEM、STORE、CUSTOMER 和 STORE_SALES 四个表，这些表的主键将在 4.2.5 小节讲述。

```
CREATE TABLE "TPCDS   "."ITEM"  (
          "I_ITEM_SK" INTEGER NOT NULL ,
          "I_ITEM_ID" CHAR(16) NOT NULL ,
          "I_REC_START_DATE" DATE ,
          "I_REC_END_DATE" DATE ,
          "I_ITEM_DESC" VARCHAR(200) ,
          "I_CURRENT_PRICE" DECIMAL(7,2) ,
          "I_WHOLESALE_COST" DECIMAL(7,2) ,
          "I_BRAND_ID" INTEGER ,
          "I_BRAND" CHAR(50) ,
          "I_CLASS_ID" INTEGER ,
          "I_CLASS" CHAR(50) ,
          "I_CATEGORY_ID" INTEGER ,
          "I_CATEGORY" CHAR(50) ,
          "I_MANUFACT_ID" INTEGER ,
          "I_MANUFACT" CHAR(50) ,
          "I_SIZE" CHAR(20) ,
          "I_FORMULATION" CHAR(20) ,
          "I_COLOR" CHAR(20) ,
          "I_UNITS" CHAR(10) ,
          "I_CONTAINER" CHAR(10) ,
          "I_MANAGER_ID" INTEGER ,
          "I_PRODUCT_NAME" CHAR(50) )
          IN "TS_PD_DATA_001" INDEX IN "TS_PD_INDX_001" ;

CREATE TABLE "TPCDS   "."STORE"  (
          "S_STORE_SK" INTEGER NOT NULL ,
          "S_STORE_ID" CHAR(16) NOT NULL ,
          "S_REC_START_DATE" DATE ,
          "S_REC_END_DATE" DATE ,
          "S_CLOSED_DATE_SK" INTEGER ,
          "S_STORE_NAME" VARCHAR(50) ,
          "S_NUMBER_EMPLOYEES" INTEGER ,
          "S_FLOOR_SPACE" INTEGER ,
          "S_HOURS" CHAR(20) ,
          "S_MANAGER" VARCHAR(40) ,
          "S_MARKET_ID" INTEGER ,
          "S_GEOGRAPHY_CLASS" VARCHAR(100) ,
          "S_MARKET_DESC" VARCHAR(100) ,
          "S_MARKET_MANAGER" VARCHAR(40) ,
          "S_DIVISION_ID" INTEGER ,
          "S_DIVISION_NAME" VARCHAR(50) ,
```

```
            "S_COMPANY_ID" INTEGER ,
            "S_COMPANY_NAME" VARCHAR(50) ,
            "S_STREET_NUMBER" VARCHAR(10) ,
            "S_STREET_NAME" VARCHAR(60) ,
            "S_STREET_TYPE" CHAR(15) ,
            "S_SUITE_NUMBER" CHAR(10) ,
            "S_CITY" VARCHAR(60) ,
            "S_COUNTY" VARCHAR(30) ,
            "S_STATE" CHAR(2) ,
            "S_ZIP" CHAR(10) ,
            "S_COUNTRY" VARCHAR(20) ,
            "S_GMT_OFFSET" DECIMAL(5,2) ,
            "S_TAX_PRECENTAGE" DECIMAL(5,2) )
           IN "TS_SM_DATA_001" INDEX IN "TS_SM_INDX_001";

CREATE TABLE "TPCDS   "."CUSTOMER"  (
            "C_CUSTOMER_SK" INTEGER NOT NULL ,
            "C_CUSTOMER_ID" CHAR(16) NOT NULL ,
            "C_CURRENT_CDEMO_SK" INTEGER ,
            "C_CURRENT_HDEMO_SK" INTEGER ,
            "C_CURRENT_ADDR_SK" INTEGER ,
            "C_FIRST_SHIPTO_DATE_SK" INTEGER ,
            "C_FIRST_SALES_DATE_SK" INTEGER ,
            "C_SALUTATION" CHAR(10) ,
            "C_FIRST_NAME" CHAR(20) ,
            "C_LAST_NAME" CHAR(30) ,
            "C_PREFERRED_CUST_FLAG" CHAR(1) ,
            "C_BIRTH_DAY" INTEGER ,
            "C_BIRTH_MONTH" INTEGER ,
            "C_BIRTH_YEAR" INTEGER ,
            "C_BIRTH_COUNTRY" VARCHAR(20) ,
            "C_LOGIN" CHAR(13) ,
            "C_EMAIL_ADDRESS" CHAR(50) ,
            "C_LAST_REVIEW_DATE" CHAR(10) )
             IN "TS_PD_DATA_001" INDEX IN "TS_PD_INDX_001" ;

CREATE TABLE "TPCDS   "."STORE_SALES"  (
            "SS_SOLD_DATE_SK" INTEGER ,
            "SS_SOLD_TIME_SK" INTEGER ,
            "SS_ITEM_SK" INTEGER NOT NULL ,
            "SS_CUSTOMER_SK" INTEGER ,
            "SS_CDEMO_SK" INTEGER ,
            "SS_HDEMO_SK" INTEGER ,
            "SS_ADDR_SK" INTEGER ,
            "SS_STORE_SK" INTEGER ,
            "SS_PROMO_SK" INTEGER ,
            "SS_TICKET_NUMBER" INTEGER NOT NULL ,
            "SS_QUANTITY" INTEGER ,
            "SS_WHOLESALE_COST" DECIMAL(7,2) ,
            "SS_LIST_PRICE" DECIMAL(7,2) ,
            "SS_SALES_PRICE" DECIMAL(7,2) ,
            "SS_EXT_DISCOUNT_AMT" DECIMAL(7,2) ,
            "SS_EXT_SALES_PRICE" DECIMAL(7,2) ,
            "SS_EXT_WHOLESALE_COST" DECIMAL(7,2) ,
            "SS_EXT_LIST_PRICE" DECIMAL(7,2) ,
            "SS_EXT_TAX" DECIMAL(7,2) ,
            "SS_COUPON_AMT" DECIMAL(7,2) ,
```

```
              "SS_NET_PAID" DECIMAL(7,2) ,
              "SS_NET_PAID_INC_TAX" DECIMAL(7,2) ,
              "SS_NET_PROFIT" DECIMAL(7,2) )
             COMPRESS YES
            PARTITION BY RANGE("SS_SOLD_DATE_SK")
            (PART "PART0" STARTING(MINVALUE) ENDING(2450814) IN "TS_PD_DATA_001" ,
             PART "PART1" STARTING(2450815) ENDING(2451179) IN "TS_PD_DATA_001" ,
             PART "PART2" STARTING(2451180) ENDING(2451544) IN "TS_PD_DATA_001" ,
             PART "PART3" STARTING(2451545) ENDING(2451910) IN "TS_PD_DATA_001" ,
             PART "PART4" STARTING(2451911) ENDING(2452275) IN "TS_PD_DATA_001" ,
             PART "PART5" STARTING(2452276) ENDING(2452640) IN "TS_PD_DATA_001" ,
             PART "PART6" STARTING(2452641) ENDING(2453005) IN "TS_PD_DATA_001" ,
             PART "PART7" STARTING(2453006) ENDING(MAXVALUE) IN "TS_PD_DATA_001" );
```

这个例子包含丰富的内容，可以作为数据库设计时建表的典范。下面解读一下其中的妙处。

1. 表空间的规划

首先需要规划好表空间。根据数据的访问特性（数据量大小和访问频率），规划不同的表空间，这对性能调优和运维将大有好处。索引与数据一般分开存储在不同的表空间。那么，数据放哪个表空间，索引放哪个表空间，热表数据放哪个表空间，大表数据放在哪个表空间，在建表前就需要做好存储规划。

注意：每条记录的最大长度不能超过页大小

每条记录的最大长度不能超过页大小。比如 pagesize 为 8K 的表空间，记录的最大长度不能超过 8K。如果表中需要存储 3 个 VARCHAR（8K）的字段，那么在页大小为 16K 的表空间中是放不下的，必须放在 32K 页大小的表空间中。

在这个例子中，大表 STORE_SALES、ITEM 和 CUSTOMER 的数据存放在表空间 TS_PD_INDX_001 中，而索引存储在表空间 TS_PD_INDX_001 中。表 STORE 的数据量小，访问频繁，数据表空间和索引表空间分别为 TS_SM_DATA_001 和 TS_SM_INDX_001，可以为之设置足够大的缓冲池，让这些热数据常驻内存。

如果表中有大对象数据，还应该用 long in 子句指定存储大对象表空间。

注意：DB2 没有用户级别的默认表空间

很多熟悉 Oracle 的用户在问，DB2 中有没有机制为某个用户（模式）建表时指定默认表空间。答案是没有！

2. 有意义的表名和字段名

表名和字段名在实际应用中非常重要。我在一次客户支持中，发现了一个表的名字叫 hkjf，刚开始根本不知道这表名是啥意思，后来问了客户，客户才告知是“航空计费”这四个字的首字母！对这种表名，没有好的文档和备注，很难猜出来。因此，建议要起有意义的表名和字段名，并在项目中采用一致的命名规则，这样更容易理解。

在这个例子中，表名分别为 ITEM、CUSTOMER、STORE 和 STORE_SALES，读者一看就知道它的作用。有人也许会问，为什么不直接用 SALES 代替 STORE_SALES 作为表名？这是

因为有另一个表 WEB_SALES 存储网上销售数据，而用 STORE_SALES 存储实体店销售数据。

除了表名，字段名也应该选择有意义的名字，这里总结一下上例中的几个可以借鉴的点：

（1）字段名能显而易见地表明存储数据的含义，例如，I_CURRENT_PRICE 和 SS_SALES_PRICE 等。

（2）由于 SQL 语言是不区分大小写的，多个单词连成的字段名可以用下划线“_”分隔。

（3）一致的后缀规范，比如 SK 或者 ID 作为键的标识。

（4）一致的字段名前缀，可以用表名的一个字母或者几个字母作为字段名前缀，如上例的“I_”、“C_”、“S_”和“SS_”。这种做法对 SQL 语句的编写大有好处，特别在多个表做连接的时候。例如，使用字段名前缀，下面这条 SQL 语句具有非常好的可读性：

```
   select avg(ss_quantity) agg1,
       avg(ss_list_price) agg2,
       avg(ss_coupon_amt) agg3,
       avg(ss_sales_price) agg4
 from store_sales, store, item
 where ss_item_sk = i_item_sk and
      ss_store_sk = s_store_sk and
      s_state in ('SD','FL', 'MI', 'LA', 'MO', 'SC')
 group by i_item_id, s_state
```

当然，这并不是说必须用上面的命名规则，用汉字的拼音缩写未尝不可，不过项目一开始就应该指定表和字段的一致性命名规则，并让大家共同遵守。

3. 所使用到的 CREATE TABLE 语句选项

CREATE TABLE 建表语句的选项有很多，这里介绍上面用到的两种：

（1）**COMPRESS：**压缩选项，启用压缩可以节省磁盘空间，并可以提高 I/O 性能和缓存池的命中率。如上例中对 STORE_SALES 表用 COMPRESS YES 子句启用压缩，关于压缩的相关内容请参考第 5 章。

（2）**分区技术：**上例中的 STORE_SALES 表用到了表分区技术，表分区技术将在第 5 章具体阐述。

小技巧：CREATE TABLE 其他重要选项

- APPEND ON：表示插入时直接添加到表的最后。对于高并发的 OLTP 应用，对目标表启用 APPEND ON 就不需要在插入的过程中寻找空闲空间了，只需要将记录加到表的末尾即可，这样可以在一定程度上提高性能。
- PCTFREE：用来维护磁盘上空闲的空间，它的缺省值是 10，如果表中有聚集索引并且有大量的插入、更新和删除操作，则该值可以调高到 20~35。
- VOLATILE：表示表的行数会急剧变化，这样优化器不用考虑表的统计信息，会尽量使用索引扫描，而不是表扫描。

最后，建表完成后可以通过 describe table 命令查看表的字段和数据类型，下面是查看表 tpcds.customer 的输出结果：

```
db2 => describe table tpcds.customer
```

```
                            Data type                      Column
Column name                  schema    Data type name      Length     Scale Nulls
---------------------------- --------- ------------------- ---------- ----- ------
C_CUSTOMER_SK                SYSIBM    INTEGER                      4     0 No
C_CUSTOMER_ID                SYSIBM    CHARACTER                   16     0 No
C_CURRENT_CDEMO_SK           SYSIBM    INTEGER                      4     0 Yes
C_CURRENT_HDEMO_SK           SYSIBM    INTEGER                      4     0 Yes
C_CURRENT_ADDR_SK            SYSIBM    INTEGER                      4     0 Yes
C_FIRST_SHIPTO_DATE_SK       SYSIBM    INTEGER                      4     0 Yes
C_FIRST_SALES_DATE_SK        SYSIBM    INTEGER                      4     0 Yes
C_SALUTATION                 SYSIBM    CHARACTER                   10     0 Yes
C_FIRST_NAME                 SYSIBM    CHARACTER                   20     0 Yes
C_LAST_NAME                  SYSIBM    CHARACTER                   30     0 Yes
C_PREFERRED_CUST_FLAG        SYSIBM    CHARACTER                    1     0 Yes
C_BIRTH_DAY                  SYSIBM    INTEGER                      4     0 Yes
C_BIRTH_MONTH                SYSIBM    INTEGER                      4     0 Yes
C_BIRTH_YEAR                 SYSIBM    INTEGER                      4     0 Yes
C_BIRTH_COUNTRY              SYSIBM    VARCHAR                     20     0 Yes
C_LOGIN                      SYSIBM    CHARACTER                   13     0 Yes
C_EMAIL_ADDRESS              SYSIBM    CHARACTER                   50     0 Yes
C_LAST_REVIEW_DATE           SYSIBM    CHARACTER                   10     0 Yes
```

4.2.5 键与约束

表约束是数据建模时结合应用需求对表中数据所做的限定，表约束对于数据的完整性有着非常重要的作用，DB2 支持的表约束有以下几种：

（1）主键约束：主键也是一种唯一性约束，是保证数据完整性的基础。一般来说，数据库的表都应该有主键。

（2）外键约束：定义两个表之间的参照完整性。引用的表称为子表，被引用的表称为父表。子表上引用的键叫外键，它引用父表上的主键。外键约束要求外键上的值在父表中必须存在。为了确保数据完整性，当对父表进行增删改等操作时，需要对子表进行相应的处理，例如级联删除、受限删除、置空值等。

（3）非空约束：限定该字段的值不能为空（即 NULL 值）。

（4）唯一性约束：要求表上这些字段的值必须唯一，不能有重复。

（5）检查约束：指定字段的值必须在检查约束的范围内。例如，限定订单表（Order）的状态（status）字段只能有五个值：CR（创建）、PY（已支付）、DR（已送达）、CN（取消）和 CO（关闭）。

1. 定义合理的主键和外键

主键是必不可少的，外键是可选的。在数据库复制的时候，也需要有主键，如果没有主键会影响复制的性能。表的主键可以采用系统生成的唯一值，也可以是用户输入，下面是 4.2.4 节用到的表 TPCDS.ITEM 和 TPCDS.STORE_SALES 的主键及其外键定义。

```
ALTER TABLE "TPCDS   "."ITEM"
    ADD PRIMARY KEY
        ("I_ITEM_SK");

ALTER TABLE "TPCDS   "."STORE_SALES"
```

```
      ADD PRIMARY KEY
          ("SS_ITEM_SK",
           "SS_TICKET_NUMBER");

ALTER TABLE "TPCDS   "."STORE_SALES"
      ADD CONSTRAINT "FK3A" FOREIGN KEY
          ("SS_ITEM_SK")
      REFERENCES "TPCDS   "."ITEM"
          ("I_ITEM_SK")
      ON DELETE NO ACTION
      ON UPDATE NO ACTION
      NOT ENFORCED
      ENABLE QUERY OPTIMIZATION;
```

2. 尽量少使用空值字段

在应用程序中，将诸如“C1 IS NOT NULL”或者“C1 IS NULL”这种条件拼接到 SQL 语句中，导致 SQL 语句复杂难读，程序代码复杂。实际上，如果一个表的字段大多可为空值，这说明数据库设计可能存在缺陷。

因此，建议尽量将字段定义为 NOT NULL，可以使用默认值代替。当然，空值的存在本身也有它的价值和含义，提倡少用并不是说不能用。

下面用一个例子来说明。如下所示，大部分字段都是 NOT NULL，只有 CHANGEREASON 和 NOTES 允许空值，从业务角度看，这两个字段确实需要空值，应该说这是一个对空值属性处理较好的表设计。

```
CREATE TABLE "DB2ADMIN"."BILLINGMETHOD"  (
              "ACTIVERECORD" CHAR(1) NOT NULL ,
              "BILLINGMETHOD" CHAR(2) NOT NULL ,
              "CHANGEREASON" CHAR(4) ,
              "DATAINTEGRITYINDICATOR" CHAR(1) NOT NULL ,
              "DATETIMESTAMP" TIMESTAMP NOT NULL WITH DEFAULT CURRENT TIMESTAMP ,
              "DEBTORREFERENCENUMBER" CHAR(10) NOT NULL ,
              "ENDDATE" TIMESTAMP NOT NULL ,
              "STARTDATE" TIMESTAMP NOT NULL ,
              "NOTES" VARCHAR(1000),
              "PROGRAMIDENTIFIER" CHAR(10) NOT NULL ,
              "SERVICEPROVIDER" CHAR(8) NOT NULL ,
              "USERIDENTIFIER" CHAR(10) NOT NULL ,
              "WORKSTATIONIDENTIFIER" CHAR(10) NOT NULL );
```

还有一种自以为灵活的方式是在表中保留一个属性字段，将各种可能为空值的数据组装保存在属性字段中，这的确避免了空值的出现。但是这种设计需要读取数据后，在 SQL 或者应用程序中进行处理，比如用 substr 或者位操作来处理数据。所以，需要根据具体情况权衡处理。

3. 显式声明约束

约束有助于控制输入值的范围，有助于保证数据完整性。对于约束规则，应该在数据库中显式声明。例如，要求数据库引擎对字符串进行算术处理时——当定义了字符串型字段来存储数字数据，而其中不慎包含非数字字符时，就会出现这样的错误。

在数据库中只需定义约束一次，就可以为应用程序使用，否则就必须由应用程序自己来处理了，所以约束在数据库上定义比较合适。

4.2.6 序列、标识列和生成列

本节探讨序列、标识列和生成列这三个最容易混淆的概念，最后讲解一下在数据迁移中需要注意的地方。

1. 序列

序列（Sequence）和标识列（Identity Column）都能自动生成唯一的数值序列，这些唯一值往往用来作为数据行的唯一标识。序列和标识列支持的类型包括 SMALLINT、INTEGER、BIGINT 和 DECIMAL（*n*，0）。

比较：序列和标识列的区别

- 标识列与特定的表绑定，而且每个表最多只能定义一个标识列。
- 序列是独立的数据库对象，不与表关联，通过 NEXTVAL 来获取下一个值，可以被多个表同时使用。

序列对象在以下情况下很有用：

（1）可以使用序列来生成键，用以跨多个表协调各键，不至于导致混乱。既简单方便，又便于管理。

（2）每个表有多个列需要自动生成的值，可以通过使用同一个序列或多个序列为每一行生成多个值。

下面这个例子创建了一个 SEQUENCE 对象，它被用来为两个表的唯一列提供生成值。

```
CREATE SEQUENCE DB2ADMIN.CHARGESEQ AS BIGINT
        START WITH 1 INCREMENT BY 1
        CACHE 10
    NO CYCLE;

INSERT INTO DB2ADMIN.CHARGE_WALTER VALUES (NEXT VAL FOR DB2ADMIN.CHARGE_SEQ, 205.8);
INSERT INTO DB2ADMIN.CHARGE_ELECTRIC VALUES (NEXT VAL FOR DB2ADMIN.CHARGE_SEQ, 324.30);
```

这里需要提的是与性能相关的 CACHE 选项。CACHE 10 表示有 10 个序列值缓存在内存中，而第 11 个值则存在编目表中。缓存序列值当然是为了性能，否则每次取下一个值都需要访问编目表，这对高负载的 OLTP 应用可能带来影响。当然，如果系统崩溃，这些缓存的值会丢失，DB2 需要从编目表中获取下一个值。因此，如果不希望由于这种情况导致不连续的序列值，请用 NO CACHE 选择。

2. 标识列

先来看标识列的用法，如下所示，表 DB2ADMIN.REVENUE 上定义了一个标识列 REVNO，起始值为 1000，每次递增 1。

```
CREATE TABLE DB2ADMIN.REVENUE(
  REVNO INTEGER GENERATED ALWAYS AS IDENTITY (
       START WITH 1000 INCREMENT BY 1),
   AMOUNT DECIMAL(12,2))
```

事实上，标识列有两种生成模式：

（1）GENERATED ALWAYS：总是为新的行生成值，不允许用户提供显式的值。

（2）GENERATED BY DEFAULT：用户可以提供显式的值，在用户未提供值时，DB2 会为该列生成值。但是，这就没法保证该列上的值具有唯一性，这会违背标识列所要求的唯一性。

下面的例子展示了 GENERATED ALWAYS 模式在插入数据时不能为标识列显式提供值，否则会抛出 SQL0798N 的错误。对于回滚的行，生成值也随之抛弃，在下一次插入数据时将生成新值。

```
INSERT INTO DB2ADMIN.REVENUE VALUES (DEFAULT, 1382.06 );   ---> 插入 (1000, 1382.06)
INSERT INTO DB2ADMIN.REVENUE (AMOUNT) VALUES (2384.96);    ---> 插入 (1001, 2384.96)
INSERT INTO DB2ADMIN.REVENUE VALUES (2000, 84927.00);      ---> 错误 SQL0798N,
COMMIT;

INSERT INTO DB2ADMIN.REVENUE (AMOUNT) VALUES (879.23);     ---> 插入 (1002, 879.23)
ROLLBACK;

INSERT INTO DB2ADMIN.REVENUE (AMOUNT) VALUES (5879.23);    ---> 插入 (1003, 5879.23)
COMMIT;
--- 结果如下
SELECT *  FROM  DB2ADMIN.REVENUE;

REVNO                AMOUT
-------------------- ---------------
              1000        1382.06
              1001        2384.96
              1003        5879.23
```

3. 生成列

生成列有 GENERATED ALWAYS 和 GENERATED BY DEFAULT 两种模式。在 DB2 中，生成列有广泛的应用，比如在 5.4 节将要讲到用生成列控制 MDC 维度列的粒度。下面来看一个生成列的简单例子，定义了一个包含生成列的表 DB2ADMIN.SALARY，在插入数据时，生成列 netSalary 基于已有列 salary 和 taxRate 根据表达式自动生成数据。在定义生成列时，不仅可以使用这种公式，还可以使用函数，比如 MONTH()等。

```
CREATE TABLE DB2ADMIN.SALARY (
  empno INTEGER,
  salary INTEGER,
  taxRate DECIMAL(5,2),
  netSalary DECIMAL(7,2) GENERATED ALWAYS AS (salary * (1 - taxRate))
);

INSERT INTO DB2ADMIN.SALARY (empno, salary, taxRate) VALUES (1001, 40000, 0.35);

SELECT * FROM DB2ADMIN.SALARY;

EMPNO      SALARY     TAXRATE NETSALARY
```

```
----------- ----------- ------- ---------
       1001      40000    0.35  26000.00
```

4. 数据迁移

对于使用生成列（包括标识列）的表，在导入数据时可能遇到下面的错误：

```
SQL3550W  在行 "F0-1" 列 "1" 中的字段值不是 NULL，但是目标列已被定义为
GENERATED ALWAYS。
```

那么怎么解决呢？这需要特别的处理。对于标识列，可以在 LOAD 时使用 identityoverride 文件类型修饰符，将这些值装入到 GENERATED ALWAYS 的标识列；对于生成列，则使用 generatedoverride 文件类型修饰符。IMPORT 也有类似的选项 identityignore 和 generatedignore，但是机制不一样，读者可以进一步了解。

在迁移数据库时，保持序列生成值的一致性很重要。如果从原始库导出 DDL 的时间与导出数据的时间不一致（就是说在这两个时间点之间数据做了修改），那么可能导致目标库的某些序列生成重复值。遇到这种情况，可以使用 DB2 提供的下列语句来修改序列的重启值，下面的语句用于修改序列 DB2ADMIN.CHARGESEQ 的重启值：

```
ALTER SEQUENCE DB2ADMIN.CHARGESEQ RESTART WITH 327691100
```

4.2.7 处理大对象的智慧

数据库需要处理的数据是多种多样的，对于大段的文本、图片、音频和视频等，DB2 通过大对象数据类型来存储和处理。如 4.2.3 节所述，DB2 中的大对象类型有 CLOB、DBCLOB 和 BLOB。CLOB 存储字符大对象，DBCLOB 存储双字节字符大对象，BLOB 存储图片和音视频等二进制内容。

1. 大对象的处理特点

LOB 数据（或 XML）不与数据行存储在一起，LOB 存储在外部存储上，每个 LOB 字段在行记录中只是存储一个 LOB 描述符，指向 LOB 数据存储的位置。

关于大对象的处理，从下面几个特点去考虑：

（1）在定义含有 LOB 或者 XML 数据的表时，应该为这些大对象的存储创建相应的表空间，然后在 CREATE TABLE 中用“LONG IN *tablespace-name*”子句指定大对象的表空间。如果没有指定大对象表空间，默认情况下，大对象会存储在该表所在的数据表空间。

（2）LOB 数据的处理一般通过应用程序处理，比如 C 程序、Java 程序等。

（3）循环日志模式下，LOB 字段的更改不会记录日志；归档日志模式下，默认对 LOB 字段的更改记录日志，但是可以用 NOT LOGGED 选项选择不记日志。

（4）对常规数据的存取，会利用缓冲池进行缓存。但是 LOB 数据的查询和存储则不通过缓冲池。为提高性能，可考虑文件系统缓存，即在创建大对象表空间指定 FILE SYSTEM CACHING 选项。可以考虑用 VARCHAR 和 VARCHAR FOR BIT 来代替不超过 32K 的大对象。

2. 内联大对象

某些应用中，并不是所有的大对象数据都很大，有些只有几百字节或者几 K。如果能把其

中某些较小的LOB数据直接存储在行记录中，会有助于性能提升，这就是内联LOB（inline LOB）的思想。

内联 LOB 是 DB2 V9.7 引入的新特性，与传统的 LOB 访问相比，可以利用缓冲池缓存，提高 LOB 数据的查询速度。同时，内联 LOB 数据也能利用压缩功能减少数据存储空间。

比较：内联 LOB 和内联 XML

在 DB2 V9.7 中，XML 也是可以内联的，它与 LOB 的内联机制是一样。区别在于，LOB 默认是内联的，XML 并不是默认内联。

内联 LOB 通过 CREATE TABLE 语句的 INLINE LENGTH 选项来指定。例如，在下面定义的表 tlob 中，CLOB 的最大长度指定为 2M，而内联长度指定为 1000 字节。这样，如果记录的长度小于 1000，那么数据将存放在基表中，其他长度大于 1000 的大对象保存在外部存储中，如图 4-2 所示。

```
CREATE TABLE tlob (a int, b char(5), c clob(2M) INLINE LENGTH 1000);
```

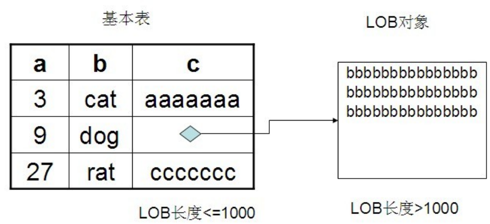


图 4-2　LOB 内联示意图

对于内联 LOB，最重要的任务是决定 INLINE LENGTH 的长度。

- 行记录的长度不能超过页面大小，也就是内联长度加上其他关系数据字段的长度和不能超过页面大小。
- 可以根据已有的样本数据来估计 LOB 数据的内联长度，这可以使用 DB2 V9.7 提供的函数 admin_est_inline_length 来估算。
- 对于内联 LOB 字段，可以使用函数 admin_is_inlined 来查询每个 LOB 是否是内联的。
- 下面讲述在 DB2 V9.7 中，如何使用 LOB 内联：

（1）首先创建 tlob 表，其中字段 c 为 CLOB（2M），并没有显式指定为内联。导入一些数据后，用 admin_is_inlined 函数查询发现，字段 c 在主键为“3”的 CLOB 数据已经是内联的，主键为“9”和“27”是非内联的。

```
db2 => CREATE TABLE tlob (a int  not null primary key, b char(5), c clob(2M));
DB20000I  The SQL command completed successfully.
# 导入一些数据
... ...
db2 => select a, admin_is_inlined(c) as IS_INLINED from tlob;

A          IS_INLINED
----------- ------
```

```
         3      1
         9      0
        27      0

  3 record(s) selected.
```

（2）查询 SYSCAT.COLUMNS，发现 CLOB 字段 c 的内联长度为 164，说明默认的内联长度相对较小。

```
    db2 => select substr(tabname,1,10) as tabname, substr(colname, 1,10) as colname,
inline_length from syscat.columns where tabname ='TLOB'

    TABNAME    COLNAME    INLINE_LENGTH
    ---------- ---------- -------------
    TLOB       A                      0
    TLOB       B                      0
    TLOB       C                    164

  3 record(s) selected.
```

（3）使用 admin_est_inline_length 函数估算这些 CLOB 数据内联所需要的存储长度，可以发现主键为“3”的 CLOB 字段内联长度为 20，小于 164，所以可以成功内联；但是，主键为“9”和“27”的 CLOB 字段内联需要的存储长度分别为 1389 和 368，都超过了 164，所以没有实现内联。

```
    db2 => select a, admin_est_inline_length(c) as EST_INLINE_LENGTH from tlob;

    A          EST_INLINE_LENGTH
    ----------- -----------
             3          20
             9         789
            27         368
  3 record(s) selected.
```

（4）为了实现更多数据的内联，可以通过 ALTER TABLE 语句将内联长度从 164 修改为 1000。然后用带有 longlobdata 选项的 reorg 命令重组表。

```
    db2 => alter table tlob alter column c set inline length 1000;
    DB20000I  The SQL command completed successfully.
    db2 => reorg table tlob longlobdata;
    DB20000I  The REORG command completed successfully.
```

注意：修改内联长度的技术要求

修改内联长度时只能增大内联长度，例如从 164 到 1000，不能减小，即不能从 1000 到 164。

（5）再次使用 admin_is_inlined 函数查询，发现本例中所有的 CLOB 数据都已经成功内联了。

```
    db2 => select a, admin_is_inlined(c) as IS_INLINED from tlob;

    A          IS_INLINED
    ----------- ------
             3      1
             9      1
```

```
        27      1

  3 record(s) selected.
```

4.3　扑朔迷离的临时表

在处理复杂业务逻辑的应用中，特别是存储过程中，临时表有着广泛的应用，主要用于存储一些中间处理结果。DB2 支持两种临时表：已声明全局临时表（DGTT）和已创建全局临时表（CGTT），它们之间的区别主要在于作用域，在下面将会具体探讨。

要想用好临时表不是一件容易的事，很多用户有这样的问题：

（1）临时表能用来在存储过程之间传递数据吗？

（2）临时表的数据在提交之后怎么就消失了？

（3）临时表能用来在不同会话（session）间传递数据吗？

这一节将深入阐述临时表的特性和使用，并回答上述问题。

注意：必须为临时表创建表空间

必须为临时表创建合适的用户临时表空间，否则 DB2 会报 SQL0286N 的错误，说没有合适的表空间，所以在数据库存储规划时，应根据需要为临时表创建相应的表空间和缓冲池。

例如，创建页大小为 16K 的用户临时表空间 ts_tmp16k，这个临时表空间使用缓冲池 bp_tmp16k:

```
CREATE BUFFERPOOL bp_tmp16k SIZE 1500 PAGESIZE 16K
CREATE USER TEMPORARY TABLESPACE ts_tmp16k pagesize 16K MANAGED BY DATABASE
 USING (FILE  '/db2data/TSTMP.DAT' 2 G )
 BUFFERPOOL bp_tmp16k
```

4.3.1　声明全局临时表（DGTT）

在 DB2 V9.5 及其以前的版本中，声明全局临时表（Declared Global Temporay Table）是唯一支持的用户自定义临时表，DGTT 的定义并不会在 DB2 的系统编目表中存储，那么 DGTT 的作用域是怎么样的呢？

（1）表定义作用域：作用域只在当前会话中，也就是说，在每个会话中都必须声明，然后才能使用它，会话结束时临时表将被删除。这种作用域只与会话有关，而与存储过程的作用域或者调用关系没有任何关系。例如，在存储过程 A 中声明了临时表 T，A 调用存储过程 B，在 B 中仍然可以使用临时表 T，甚至调用 A 的程序也可以操作临时表 T。

注意：DGTT 只与会话相关

DGTT 的模式名必须为 SESSION。只要同一个会话中，临时表就可以被所有的存储过程访问，也不需要用参数来传递临时表。

（2）数据作用域：临时表中的数据是会话隔离的，不同会话之间不会共享数据。这又可以细分为如下两种：

- 事务级：在声明时由 ON COMMIT DELETE ROWS 子句指定，每次事务提交（或回滚）后，DB2 将删除临时表中的数据。
- 会话级：在声明时由 ON COMMIT PRESERVE ROWS 子句指定，事务提交时临时表的数据将保留，数据在整个会话期间都存在，会话结束时数据才被清除。

还可以使用 NOT LOGGED 子句指定对 DGTT 临时表的操作不记录日志，创建 DGTT 的语法使用“DECLARE GLOBAL TEMPORARY TABLE”，如下所示，读者需要注意其中的 ON COMMIT [PRESERVE|DELETE] ROWS 选项，如下所示：

```
DECLARE GLOBAL TEMPORARY TABLE <table_name> (
<columns definition> )
ON COMMIT [PRESERVE|DELETE] ROWS
[NOT LOGGED ON ROLLBACK [PRESERVE|DELETE] ROWS ]
[WITH REPLACE]
IN <tablespace-name>;
```

下面的例子声明了临时表 session.dgtt1，并指定了 ON COMMIT DELETE 选项。插入 2 条记录后，能查到新插入的数据。事务提交后，数据被自动删除。会话结束后，临时表将不复存在。

```
CONNECT TO dbname;
update command options using c off;  --关闭自动提交开关

DECLARE GLOBAL TEMPORARY TABLE session.dgtt1
(id INT NOT NULL, name VARCHAR(20), d DOUBLE)
 ON COMMIT DELETE ROWS IN ts_tmp16k;

INSERT INTO session.dgtt1(id, name, d)
   VALUES (1, 'Jim', 3459.67), (2, 'Tom', 105.7);
DB20000I  The SQL command completed successfully.

SELECT * FROM session.dgtt1

ID          NAME                 D
----------- -------------------- ------------------------
          1 Jim                  +3.45967000000000E+003
          2 Tom                  +1.05700000000000E+002

  2 record(s) selected.

COMMIT;
DB20000I  The SQL command completed successfully.

SELECT * FROM session.dgtt1

ID          NAME                 D
----------- -------------------- ------------------------

  0 record(s) selected.
```

下面的例子声明了临时表 session.dgtt1，并指定了 ON COMMIT PRESERVE 选项。插入 2 条记录后，能查到新插入的数据。事务提交后，数据依然保留，仍能查询到这些数据。会话结束后，临时表也不复存在。

```
DECLARE GLOBAL TEMPORARY TABLE session.dgtt1
   (id INT NOT NULL, name VARCHAR(20), d DOUBLE)
   ON COMMIT PRESERVE ROWS IN ts_tmp16k;

INSERT INTO session.dgtt1(id, name, d)
   VALUES (1, 'Jim', 3459.67), (2, 'Tom', 105.7);
COMMIT;
SELECT * FROM session.dgtt1

ID          NAME                 D
----------- -------------------- ------------------------
          1 Jim                   +3.45967000000000E+003
          2 Tom                   +1.05700000000000E+002

  2 record(s) selected.

CONNECT RESET;
CONNECT TO dbname;
SELECT * FROM session.dgtt1
SQL0204N  "SESSION.DGTT1" is an undefined name.  SQLSTATE=42704
```

4.3.2　创建全局临时表（CGTT）

DB2 V9.7 开始支持创建全局临时表（CGTT），为了帮助读者更好地理解 CGTT，接下来通过和 DGTT 对比的方式探讨一下 CGTT。

- 表定义作用域，与 DGTT 不同，CGTT 的定义持久保存在 DB2 的系统编目中，不需要在每个会话中重新定义，而是在各个会话之间共享。
- 数据作用域：与 DGTT 基本类似，CGTT 数据在会话中是私有的，在会话之间隔离，每个会话操作的数据只能在本会话中可见，其他会话的数据在本会话不可见（仿佛那些数据不存在），会话结束后，该会话操作的数据将被清除。

从下面的存储过程可以看到，创建 CGTT 使用“CREATE GLOBAL TEMPORARY TABLE”语句，由于表定义在不同会话之间共享，存储过程 sp_a 和 sp_b 都可以操作预先定义的 db2admin.cgtt。

```
CREATE GLOBAL TEMPORARY TABLE db2admin.cgtt
   (id INT NOT NULL, name VARCHAR(20), v INT)
   ON COMMIT DELETE ROWS IN ts_tmp16k@

CREATE or replace PROCEDURE sp_b(in uid int, in ratio INT)
  BEGIN
     UPDATE db2admin.cgtt SET v = v * ratio WHERE id = uid;
  END@

CREATE or replace PROCEDURE sp_a(out sum INT)
  BEGIN
```

```
    INSERT INTO db2admin.cgtt(id, name, v)
      VALUES (1, 'Jim', 100), (2, 'Tom', 200);
    CALL sp_b(1, 3);
    SELECT SUM(v) INTO sum FROM db2admin.cgtt;
 END@

CALL sp_a(?)

  Value of output parameters
  --------------------------
  Parameter Name  : SUM
  Parameter Value : 500

  Return Status = 0
```

如果用 DGTT 实现这样的逻辑，则稍微麻烦点。具体的实现代码如下所示，在 sp_a 中声明 DGTT session.tmp1，然后在 sp_b 中用动态 SQL 语句修改 session.tmp1 中的数据。

```
CREATE or replace PROCEDURE sp_a(out sum INT)
  begin
    declare global temporary table session.tmp1
      (id INT NOT NULL, name VARCHAR(20), v INT);
    INSERT INTO session.tmp1 (id, name, v)
      VALUES (1, 'Jim', 100), (2, 'Tom', 200);
    CALL sp_b(1, 3);
    SELECT SUM(v) INTO sum FROM session.tmp1;
 end@
CREATE or replace PROCEDURE sp_b(in uid int, in ratio INT)
  begin
     EXECUTE IMMEDIATE 'UPDATE session.tmp1 SET v = v * '
           || ratio || ' WHERE id = ' || uid;
  end@
```

注意：只能使用动态 SQL 语句访问 session.tmp1

在 sp_b 中不能使用静态 SQL 语句访问 session.tmp1，这是因为 session.tmp1 是在执行 sp_a 时声明的，在创建 sp_b 时编译器无法确定 session.tmp1 是否已声明。

4.3.3 临时表小结

在 DB2 V9.7 中，如果临时表的数据量比较大，可以在临时表上创建索引，也可以对临时表启用压缩。

普通表相比，临时表（包括 DGTT 和 CGTT）有如下优点：

- 由于临时表中的数据在各会话之间隔离，不需要行锁定，所以能提高并发性能。
- 在某些应用中可使用临时表 NOT LOGGED 选项，不记录日志，从而提高性能。
- 如果临时表创建了索引，DB2 编译器会将这些索引考虑在内。由于特定会话中的 Runstats 只会考虑本地数据，这将允许各会话使用不同的访问计划。

临时表在许多应用下都可以带来好处。最容易理解的应用是在线购物车，会话开始后每个

用户可以往购物车加商品，只有在用户真正购买商品后才会复制到实际的订单表中，并在会话结束时销毁它的数据。请参考《从 Oracle 到 DB2 开发从容转身》第 5 章相关内容。

4.4　无处不在的时态表

“时间就是金钱，效率就是生命。”DB2 V10 新引入的时态数据管理技术以一种简洁有效的方式管理随时间变化的数据版本，这可以节省数据库管理员和应用开发人员大量的时间和精力。

时态数据几乎在所有行业都有着广泛的应用场景，比如银行、保险、金融、零售和电子商务等。

4.4.1　无处不在的时间

数据随时间而变化，先来看一下时态数据的具体应用场景：

（1）审计人员对金融机构的交易员在过去 1 年内的交易记录进行审计。

（2）证券公司记录股民的详细交易记录，用于审计和用户查询分析。

（3）某在线购物网站对用户消费的历史数据进行分析，并有针对性地推出某些产品的打折消息。

（4）保险公司确认发生事故后，根据客户保单的时间周期和有效条款进行赔偿。

（5）项目管理中，记录人员在各个项目中的工作时间，从而计算工时和评估绩效，比如 A 员工从 2012-03-04 到 2012-04-08 在甲项目中工作，而 2012-04-09 到 2012-03-15 在乙项目中工作。

（6）一家在线旅行公司为客户规划 10 天的旅游行程，并希望检测行程中的不一致性。比如预定北京一家酒店 3 天，但其中的一天却同时预订了上海的一家酒店，这种冲突需要避免。

（7）信用卡用户查询消费记录，如果发现交易的数据错误或者利率计算有问题，银行需要追溯以前的数据来更正错误或重新计算利息。

在 DB2 V10 以前的版本中，实现这样的时态数据管理，并不是件轻松的事情，需要创建触发器或复杂的应用逻辑来管理时态数据。DB2 V10 引入了时态表，简化了应用的处理逻辑，实现了数据版本的自动管理。

比较：系统时间和应用时间

系统时间跟踪何时对表的数据进行了更改，比如股票交易的时间、活期银行账户的存取款时间；应用时间也称为业务时间，用来跟踪业务条件的有效期。比如保单的有效时间周期或定期贷款的期限。换句话说，系统时间就是系统自动生成的时间，而应用时间则是用户指定的时间。

场景（1）、（2）和（3）对应的是系统时间，它指的是数据的历史版本；场景（4）、（5）和（6）需要的是应用时间，它管理各种业务条件的有效期；场景（7）则同时需要系统时间和应用时间。

根据处理时间的方式，可以将时态表分成下面的 3 种类型：

- 系统时间周期的时态表（System-period Temporal Table，简写为 STT）。
- 应用时间周期的时态表（Application-period Temporal Table，简写为 ATT）。
- 双时态表（Bitemporal Table）：结合了 STT 和 ATT 两种特性，既拥有业务时间周期特性，又拥有系统时间周期特性。

对于时态表，不管是系统时态表还是应用时态表，都有额外的两个时间字段，即起始时间和结束时间用来表明每一行数据的有效时间周期。对于系统时态表，还有一个相同结构的历史表与之关联，用于存储历史数据版本，当系统时态表中的数据被更新时，DB2 自动将更新数据转移到历史表中。

4.4.2 系统时态表

下面通过股票交易的例子来说明，这个例子包括股票的买入、增持和卖出等操作。在处理查询时，DB2 会根据需要透明地访问历史表，从而轻松处理历史数据，并不需要应用编写包含各种时间戳的复杂 WHERE 子句。

1. 创建系统时态表

包括下面 3 个步骤：

（1）创建系统时态表 stock

除了基本字段外，基表还需要包含额外的 3 个 TIMESTAMP(12) 字段，这 3 个字段可定义为 GENERATED ALWAYS，这样在插入、更新和删除时，DB2 就会自动生成这 3 个字段的值。

如下所示，创建时态表 stock，其中 sys_start 为开始时间，sys_end 为结束时间，PERIOD SYSTEM_TIME(sys_start,sys_end)子句指定数据的有效时间段。trans_start 为影响当前行数据变化的事务开始时间，将其指定为隐藏字段（IMPLICITLY HIDDEN）。

```
CREATE TABLE stock (
  account_no    INTEGER NOT NULL,
  stock_no      INTEGER NOT NULL,
  shares        DECIMAL(15,2) NOT NULL,
  sys_start     TIMESTAMP(12) NOT NULL
                 GENERATED ALWAYS AS ROW BEGIN,
  sys_end       TIMESTAMP(12) NOT NULL
                 GENERATED ALWAYS AS ROW END,
  trans_start  TIMESTAMP(12)
                 GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN,
    PERIOD SYSTEM_TIME (sys_start, sys_end)
);
```

（2）创建历史表 stock_history

历史表的定义结构与当前表相同，可使用 CREATE TABLE…LIKE 语句轻松创建。

```
CREATE TABLE stock_history LIKE stock ;
```

（3）关联历史表，启用版本控制

使用 ALTER TABLE 语句的 ADD VERSIONING 子句将历史表关联到当前表并启用版本控制。

```
    ALTER TABLE stock
       ADD VERSIONING USE HISTORY TABLE stock_history ;
```

2. 对系统时态表插入、更新和删除操作

先查看一下当前表 stock 和历史表 stock_history 中的数据：

```
    select * from stock

    ACCOUNT_NO  STOCK_NO   SHARES          SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
          12001      600068         5500.00 2012-12-01-09.40.32.000000000000
9999-12-30-00.00.00.000000000000
          12001      600735        12000.00 2012-10-15-14.48.02.000000000000
9999-12-30-00.00.00.000000000000
          12001      601633         7500.00 2011-09-09-10.25.28.000000000000
9999-12-30-00.00.00.000000000000
          12001      602179        10000.00 2013-01-28-13.21.18.000000000000
9999-12-30-00.00.00.000000000000

      4 record(s) selected.

    select * from stock_history

    ACCOUNT_NO  STOCK_NO   SHARES          SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
          12001      600068         2500.00 2011-09-01-14.00.00.000000000000
2012-12-01-09.40.32.000000000000
          12001      600731         8000.00 2009-12-01-10.03.18.000000000000
2010-01-01-13.21.18.000000000000
          12001      602179        10000.00 2010-02-01-05.45.02.000000000000
2013-01-28-13.21.18.000000000000

      3 record(s) selected.
```

注意：

定义为 IMPLICITLY HIDDEN 的 trans_start 字段在“SELECT *”语句中并不会显示。

（1）插入时态数据

插入时态数据时无须对 3 个时间字段赋值，DB2 会自动为它们生成合适的值。如下所示，sys_start 的值被设置为开始时间，而 sys_end 被值设置为最大值 9999-12-30-00.00.00.000000000000。

```
    INSERT INTO stock (account_no, stock_no, shares) VALUES (12001, 608712, 1500)
    DB20000I  The SQL command completed successfully.

    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock WHERE account_no
= 12001 and stock_no = 608712
```

```
    ACCOUNT_NO  STOCK_NO    SHARES          SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      608712         1500.00 2013-02-03-02.15.44.886174000000
9999-12-30-00.00.00.000000000000

      1 record(s) selected.
```

（2）更新时态数据

更新时态数据时，会产生数据版本变化，更新前的版本会自动保存到历史表中。DB2 将历史表中的 sys_end 值和当前表中的 sys_start 值设置为该事务的开始时间。如果在同一事务内多次更新行，DB2 并不会在历史表创建多条历史记录，而仅仅保留最初的旧版本。

```
    UPDATE stock
      SET shares = shares + 3000
      WHERE account_no = 12001 and  stock_no = 600068;

    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock WHERE account_no
= 12001 and  stock_no = 600068

    ACCOUNT_NO  STOCK_NO    SHARES          SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      600068         8500.00 2013-02-03-02.15.45.080586000000
9999-12-30-00.00.00.000000000000

      1 record(s) selected.

    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock_history WHERE
account_no = 12001 and  stock_no = 600068

    ACCOUNT_NO  STOCK_NO    SHARES          SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      600068         2500.00 2011-09-01-14.00.00.000000000000
2012-12-01-09.40.32.000000000000
         12001      600068         5500.00 2012-12-01-09.40.32.000000000000
2013-02-03-02.15.45.080586000000

      2 record(s) selected.
```

（3）删除系统时态数据

删除数据时，DB2 会将当前表中的数据删除，并将删除的数据版本自动保存到历史表中。历史表的 sys_end 值设置为该事务的开始时间。

```
    DELETE FROM stock WHERE account_no = 12001 and  stock_no = 600735
    DB20000I  The SQL command completed successfully.

    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock_history WHERE
account_no = 12001 and  stock_no = 600735
```

```
    ACCOUNT_NO  STOCK_NO    SHARES           SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      600735         12000.00 2012-10-15-14.48.02.000000000000
2013-02-03-02.15.45.375786000000

      1 record(s) selected.
```

3. 查询系统时态数据

查询系统时态表的当前数据与查询普通表完全一样，得到的数据为最新版本。如果需要查询各种历史版本，可以使用 DB2 提供的 3 种新语法：

- FOR SYSTEM_TIME AS OF timestamp：查询某个时间点的数据，DB2 会自动访问历史表获取合适的数据。
- FOR SYSTEM_TIME FROM begin TO end：查询时间点从 begin 到 end 的数据版本。该语法是半开区间，“含头不含尾”，即指定的 begin 包含在时间段内，但 end 不包含在内。
- FOR SYSTEM_TIME BETWEEN begin AND end：查询时间点从 begin 到 end 的数据版本。该语法是闭区间，“含头含尾”，指定的开始和结束时间都包含在时间段内。
- 使用时态条件查询系统时态表的示例如下所示：

```
    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock
       FOR SYSTEM_TIME AS OF '2011-10-01'
       WHERE account_no = 12001

    ACCOUNT_NO  STOCK_NO    SHARES           SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      600068          2500.00 2011-09-01-14.00.00.000000000000
2012-12-01-09.40.32.000000000000
         12001      602179         10000.00 2010-02-01-05.45.02.000000000000
2013-01-28-13.21.18.000000000000
         12001      601633          7500.00 2011-09-09-10.25.28.000000000000
9999-12-30-00.00.00.000000000000

      3 record(s) selected.

    SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock
       FOR SYSTEM_TIME FROM '2009-11-01-00.00.00' TO '2011-09-01-14.00.00'
       WHERE account_no = 12001

    ACCOUNT_NO  STOCK_NO    SHARES           SYS_START                         SYS_END
    ----------- ----------- ----------------- --------------------------------
--------------------------------
         12001      600731          8000.00 2009-12-01-10.03.18.000000000000
2010-01-01-13.21.18.000000000000
         12001      602179         10000.00 2010-02-01-05.45.02.000000000000
2013-01-28-13.21.18.000000000000
```

```
    2 record(s) selected.

  SELECT account_no, stock_no, shares, sys_start, sys_end FROM stock
     FOR SYSTEM_TIME BETWEEN '2009-11-01-00.00.00' AND '2011-09-01-14.00.00'
     WHERE account_no = 12001

  ACCOUNT_NO  STOCK_NO    SHARES           SYS_START                        SYS_END
  ----------- ----------- ----------------- --------------------------------
--------------------------------
        12001      600068         2500.00 2011-09-01-14.00.00.000000000000
2012-12-01-09.40.32.000000000000
        12001      600731         8000.00 2009-12-01-10.03.18.000000000000
2010-01-01-13.21.18.000000000000
        12001      602179        10000.00 2010-02-01-05.45.02.000000000000
2013-01-28-13.21.18.000000000000

    3 record(s) selected.
```

注意：

对时态数据的查询在 FROM 子句中仅引用当前表，DB2 会自动访问历史表。

可以看到，系统时态表的语法简单明了，极大地减少了应用逻辑，并确保所有应用程序一致地处理时间相关的数据。

4.4.3 应用时态表

应用时态表用于跟踪某些业务时间相关的数据。例如，一份保险的有效期为一年，从 2012 年 1 月 1 日到 2012 年 12 月 31 日，以及 4.4.1 小节提到的项目时间管理、旅游行程管理等。

跟系统时态表一样，应用时态表也需要使用一个时间段（业务条件的开始和结束时间点）。但是，应用时态表有下面的特点：

- 应用时态表没有独立的历史表，它的数据以及相应的业务有效时间都在一个表中进行维护。
- 用户在向应用时态表中插入数据时需要提供业务时间段的开始和结束值。
- 应用时态表不需要使用事务开始时间字段。
- 下面以保险行业的保单为例，阐述一下应用时态表的各种功能。

1. 创建应用时态表

创建应用时态表的语法很简单，例如在保单表中，bus_start 和 bus_end 字段定义为 DATE 类型，PERIOD BUSINESS_TIME 子句表明这两个字段用来跟踪每行数据的有效开始时间和结束时间。

DB2 会自动生成一个隐式约束来强制要求 bus_start 值小于 bus_end 值，从而确保时态数据的完整性。另外，在应用时态表上，还可以根据应用时间建立唯一索引，这样可以确保同一行数据不能同时拥有两个版本，如下例中的索引 idx_insr 所示：

```
  CREATE TABLE insur_policy(
    acc_no          INTEGER NOT NULL,
```

```
  insur_type        VARCHAR(10),
  insur_premium      DECIMAL(15,2) NOT NULL,
  insur_coverage     DECIMAL(15,2) NOT NULL,
  bus_start         DATE NOT NULL,
  bus_end           DATE NOT NULL,
    PERIOD BUSINESS_TIME(bus_start, bus_end)
);
CREATE UNIQUE INDEX idx_insr
  ON insur_policy(acc_no, BUSINESS_TIME WITHOUT OVERLAPS) ;
```

2. 对应用时态表插入、更新和删除操作

应用时态表的插入、更新和删除操作会引起数据业务时间有效期的变化，下面分别讲解。

（1）插入时态数据

插入数据与普通表一样，非常简单，只需为表中的字段（包括开始和结束时间字段）提供合适的值。例如，向保单表中插入数据的语句如下所示：

```
INSERT INTO insur_policy (acc_no, insur_type, insur_premium,
                       insur_coverage, bus_start, bus_end)
  VALUES (1001, 'A01', 1200, 200000, '2009-11-01', '2013-11-01'),
         (1002, 'A02', 2000, 450000, '2010-01-01', '2013-01-01'),
         (1003, 'A01', 800,  140000, '2010-02-01', '2011-02-01'),
         (1004, 'B02', 6000, 800000, '2011-01-01', '2011-12-31');

SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end FROM
insur_policy;

ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
----------- ---------- ----------------- ----------------- ---------- ----------
      1001 A01                1200.00        200000.00 11/01/2009 11/01/2013
      1002 A02                2000.00        450000.00 01/01/2010 01/01/2013
      1003 A01                 800.00        140000.00 02/01/2010 02/01/2011
      1004 B02                6000.00        800000.00 01/01/2011 12/31/2011

  4 record(s) selected.
```

注意：

与系统时态表不同，向应用时态表插入数据需要为两个日期字段提供值。

由于定义了不允许应用时间段重叠的唯一索引，向应用时态表插入时间段重叠的数据将会报错。如下所示，帐号 1002 的有效业务时间段为 2010-01-01 到 2013-01-01，在向该表插入业务时间段为 2012-01-01 到 2012-07-01 的数据时，因为违背了唯一性约束所以会报错。如果希望调整帐号 1002 在 2012-01-01 到 2012-07-01 期间的保险，可以使用合适的 UPDATE 语句来实现。

```
INSERT INTO insur_policy (acc_no, insur_type, insur_premium,
                       insur_coverage, bus_start, bus_end)
  VALUES (1002, 'A01', 1800, 300000, '2012-01-01', '2012-07-01') ;
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0803N  One or more values in the INSERT statement, UPDATE statement, or
```

```
foreign key update caused by a DELETE statement are not valid because the
primary key, unique constraint or unique index identified by "1" constrains
table "DB2ADMIN.INSUR_POLICY" from having duplicate values for the index key.
SQLSTATE=23505
```

（2）更新时态数据

对应用时态表的更新操作可以是普通的 UPDATE 语句，也可以用新语法 PORTION OF BUSINESS_TIME 子句，表示只更新某个时间段的数据。

注意：

语句中的时态限制（FOR PORTION OF BUSINESS_TIME FROM…TO…）直接跟在表名之后，而不是 WHERE 子句的一部分。

更新应用时态数据时，会产生数据有效期的变化，甚至一行数据可能被拆分成多行数据。如下所示的 UPDATE 语句提高帐号 1001 在 2010 年整个年度的保费和保额，于是原来的一条数据拆分成了 3 条数据，业务时间的有效期也相应发生了改变。

```
UPDATE insur_policy
  FOR PORTION OF BUSINESS_TIME FROM '2010-01-01' TO '2011-01-01'
  SET insur_premium = 2000, insur_coverage = 400000
  WHERE acc_no = 1001 ;

SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end
  FROM insur_policy
  WHERE acc_no = 1001
  ORDER BY acc_no, bus_start;
ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
----------- ---------- ----------------- ----------------- ---------- ----------
       1001 A01                  1200.00         200000.00 11/01/2009 01/01/2010
       1001 A01                  2000.00         400000.00 01/01/2010 01/01/2011
       1001 A01                  1200.00         200000.00 01/01/2011 11/01/2013

  3 record(s) selected.
```

注意：

这里时间段的语义仍然是半开区间，“含头不含尾”，即 bus_start 包含在有效业务时间段内，但 bus_end 不包含在内。

（3）删除应用时态数据

删除应用时态表的数据与更新操作类似，可以用普通的 DELETE 语句，也可以用 FOR PORTION OF BUSINESS_TIME 子句限定删除操作的时间范围。

如下所示的 DELETE 语句取消帐号 1001 在 2010 年 7 月 1 日到 2011 年 7 月 1 日的保险，从结果中可以看到，相应时间段的业务数据已经被清除。

```
DELETE FROM insur_policy
  FOR PORTION OF BUSINESS_TIME FROM '2010-07-01' TO '2011-07-01'
  WHERE acc_no = 1001 ;
```

```
    SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end
     FROM insur_policy
     WHERE acc_no = 1001
     ORDER BY acc_no, bus_start ;

    ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
    ----------- ---------- ----------------- ----------------- ---------- ----------
          1001 A01                 1200.00         200000.00 11/01/2009 01/01/2010
          1001 A01                 2000.00         400000.00 01/01/2010 07/01/2010
          1001 A01                 1200.00         200000.00 07/01/2011 11/01/2013

      3 record(s) selected.
```

3. 查询应用时态数据

对应用时态表，可以使用基本的 SELECT 语句，也可以使用新的时态条件谓词语法，DB2 提供了下面 3 种新的时态语法：

（1）FOR BUSINESS_TIME AS OF…个时间点的业务数据。

（2）FOR BUSINESS_TIME FROM *begin* TO *end*：查询 zz。

（3）FOR BUSINESS_TIME BETWEEN *begin* AND *end*：查询时间点从 *begin* 到 *end* 的有效业务数据。该语法是闭区间，“含头含尾”，指定的开始和结束时间都包含在时间段内。

使用时态条件查询应用时态表的例子如下所示：

```
    SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end FROM
insur_policy

    ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
    ----------- ---------- ----------------- ----------------- ---------- ----------
          1001 A01                 2000.00         400000.00 01/01/2010 07/01/2010
          1002 A02                 2000.00         450000.00 01/01/2010 01/01/2013
          1003 A01                  800.00        140000.00 02/01/2010 02/01/2011
          1004 B02                 6000.00         800000.00 01/01/2011 12/31/2011
          1001 A01                 1200.00         200000.00 11/01/2009 01/01/2010
          1001 A01                 1200.00         200000.00 07/01/2011 11/01/2013

      6 record(s) selected.

    SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end FROM
insur_policy FOR BUSINESS_TIME AS OF '2010-01-01'

    ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
    ----------- ---------- ----------------- ----------------- ---------- ----------
          1001 A01                 2000.00         400000.00 01/01/2010 07/01/2010
          1002 A02                 2000.00         450000.00 01/01/2010 01/01/2013

      2 record(s) selected.

    SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end FROM
insur_policy FOR BUSINESS_TIME FROM '2009-06-01' TO '2010-02-01'

    ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
```

```
----------- ---------- ----------------- ----------------- ---------- ----------
      1001 A01               2000.00        400000.00 01/01/2010 07/01/2010
      1002 A02               2000.00        450000.00 01/01/2010 01/01/2013
      1001 A01               1200.00        200000.00 11/01/2009 01/01/2010

  3 record(s) selected.

SELECT acc_no, insur_type, insur_premium, insur_coverage, bus_start, bus_end FROM
insur_policy FOR BUSINESS_TIME BETWEEN '2009-06-01' AND '2010-02-01'

ACC_NO      INSUR_TYPE INSUR_PREMIUM     INSUR_COVERAGE    BUS_START  BUS_END
----------- ---------- ----------------- ----------------- ---------- ----------
      1001 A01               2000.00        400000.00 01/01/2010 07/01/2010
      1002 A02               2000.00        450000.00 01/01/2010 01/01/2013
      1003 A01                800.00        140000.00 02/01/2010 02/01/2011
      1001 A01               1200.00        200000.00 11/01/2009 01/01/2010

  4 record(s) selected.
```

4.4.4 双时态表

DB2 还支持双时态表，它将二者的优势结合在一起。例如，使用应用时间来管理保单的有效期，使用系统时间来跟踪事务对这些保险单所做更改的历史记录和时间戳；使用应用时间来管理贷款的期限，使用系统时间来跟踪贷款帐号所做更改的历史记录和时间。

创建双时态表时需包括系统时态表需要的 3 个时间字段和应用时态表需要的 2 个时间字段，同时建立相应的历史表并与基础表关联，这样，通过双时态表可以轻松管理系统时间和应用时间。例如，为贷款帐户创建双时态表的语句如下所示：

```
CREATE TABLE loan_account(
  account_no    INTEGER NOT NULL,
  loan_type         VARCHAR(10),
  interest          DECIMAL(5,2) NOT NULL,
  balance           DECIMAL (10,2),
  bus_start         DATE NOT NULL,
  bus_end           DATE NOT NULL,
  sys_start         TIMESTAMP(12) NOT NULL
                     GENERATED ALWAYS AS ROW BEGIN IMPLICITLY HIDDEN,
  sys_end           TIMESTAMP(12) NOT NULL
                     GENERATED ALWAYS AS ROW END IMPLICITLY HIDDEN,
  trans_start       TIMESTAMP(12)
                    GENERATED ALWAYS AS TRANSACTION START ID IMPLICITLY HIDDEN,
    PERIOD SYSTEM_TIME (sys_start, sys_end),
    PERIOD BUSINESS_TIME (bus_start, bus_end),
    PRIMARY KEY (account_no, BUSINESS_TIME WITHOUT OVERLAPS)
) ;

CREATE TABLE loan_account_history
  LIKE loan_account;

ALTER TABLE loan_account
  ADD VERSIONING USE HISTORY TABLE loan_account_history ;
```

双时态表的操作与查询就是系统时态表和应用时态表的结合，这里不再赘述。

4.4.5　将普通表转换为时态表

如果一个表是普通表，现在希望把它转换为时态表，如何实现呢？其实，不论转换为系统时态表还是应用时态表，转换的方法都是类似的，而且由于应用时态表没有历史表，所以转换更容易些。

根据目标表是否有时间字段，可以分为以下两种情况：

（1）如果现有的表没有时间字段，那么可以使用 ALTER TABLE 语句向该表添加必需的时间字段、时间段和历史表，并将历史表和基本表关联。

（2）如果已经有了一套实现版本控制的机制，即基本表已包含时间字段并定义了相应的历史表，对基本表的版本控制逻辑由触发器实现。这时，只需执行 ALTER TABLE 语句基于现有的两个时间字段增加一个 SYSTEM_TIME 时间段的定义，保留历史表，并用 ALTER TABLE 语句将历史表和基本表关联，最后删除原来的触发器。

接下来，通过一个例子来说明一下第（2）种情况下的具体转换步骤，为了方便起见，触发器相关内容省略。下面是表 stock 和历史表 stock_history 的 DDL 定义，都已经有了两个时间字段，分别为 sys_start 和 sys_end：

```
CREATE TABLE stock (
  account_no     INTEGER NOT NULL,
  stock_no       INTEGER NOT NULL,
  shares         DECIMAL(15,2) NOT NULL,
  sys_start   TIMESTAMP(12) NOT NULL,
  sys_end     TIMESTAMP(12) NOT NULL
) ;

CREATE TABLE stock_history LIKE stock ;
```

首先，用 ALTER TABLE 语句将 stock 表中的 sys_start 和 sys_end 这两个已有的时间字段修改为“GENERATED AS ROW BEGIN”和“GENERATED AS ROW END”，并基于这两个时间字段定义一个 SYSTEM_TIME 时间段。然后，在 stock 表中添加 trans_start 字段记录当前行数据变化的事务开始时间，相应的，在历史表 stock_history 中也添加该字段。

```
ALTER TABLE stock ALTER COLUMN sys_start
  SET GENERATED AS ROW BEGIN ;

ALTER TABLE stock ALTER COLUMN sys_end
  SET GENERATED AS ROW END ;

ALTER TABLE stock
  ADD PERIOD SYSTEM_TIME (sys_start, sys_end) ;

ALTER TABLE stock ADD COLUMN trans_start TIMESTAMP(12)
  GENERATED AS TRANSACTION START ID
  IMPLICITLY HIDDEN ;

ALTER TABLE stock_history
```

```
    ADD COLUMN trans_start TIMESTAMP(12)
    IMPLICITLY HIDDEN ;
```

最后，使用 ALTER TABLE 语句将历史表 stock_history 和基本表 stock 关联，从而启用版本控制。

```
  ALTER TABLE stock
    ADD VERSIONING USE HISTORY TABLE stock_history ;
```

这样，就完成了一个从普通表到系统时态表的转换。

4.5 索引设计是一招鲜，吃遍天

当问起一些从事性能优化工作的朋友，SQL 性能优化中有哪些重要的因素？许多人首推索引。毫无疑问，索引对数据库性能有重要影响！索引对性能有这么重要的影响，那么索引如何实现？这应该是大家都想了解的一个问题。

小故事：性能优化是面向最终用户的，不是给领导看的

2010 年春节刚过，本人就被某移动客户告知，有个重要领导要访问他们，所以需要先对他们的数据库系统进行优化。于是，在重要领导访问的前夕，我来到了美丽的杭州，去从事秘密的地下优化工作。

记得就是为一个频繁调用的 SQL 语句创建了索引，现在已经记不清楚细节了，竟然取得了明显的效果：前端应用的响应时间从 20s 下降为 5s！

客户得知效果后非常开心，这下重要领导视察的时候就会非常满意了。但是，我这里需要提醒广大 DBA 的是：性能优化是面向最终用户的，不是给领导看的。

4.5.1 DB2 索引长什么样

索引其实很简单！很好理解。

索引好比书的目录。试想一下在图书馆查资料时，如果不能按照目录检索，我们将不得不翻阅所有的书籍，那么必将陷入浩瀚的文字海洋中。与此类似的，如果没有合理的索引，那么 DB2 引擎将不得不扫描全表，这意味着资源和时间的大量消耗。

那么，在 DB2 中，索引是如何实现的？答案是以 B+树为基础实现的。

B+树是用来建立索引的一种基本形式，它的叶子节点上指向记录的一系列指针可以起到迅速找到数据的作用。如图 4-3 所示，B+树能自动保持与数据文件大小相适应的索引层次，其节点分为三类，根节点、中间节点和叶子节点。中间节点用于索引数据的定位，叶子节点指向实际的数据行。

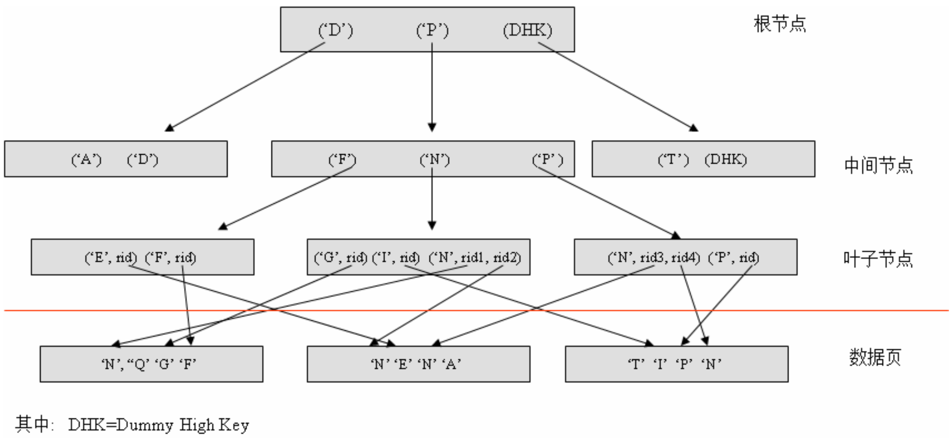


图 4-3　DB2 B+树索引

对于给定的键值，在 B+树查找记录是从根节点一直访问到叶节点，然后找到指向记录的地址（RID），时间复杂度为 O(LOG(n))，n 为键值的数目。

数据行的地址都存储在叶子节点。如图 4-4 所示，叶子页的头部有一个槽目录，槽目录的每一个条目都是一个索引项，它保存了该项的键值以及相应的记录 RID。右边的页为数据页，每个数据页开头都有一个页头。在页头后面，也有一个槽目录，槽目录的每一个条目都与该页中的一个记录相对应。该条目本身是数据页中记录开始位置的字节位移，值为－1 的条目表示已删除的记录。

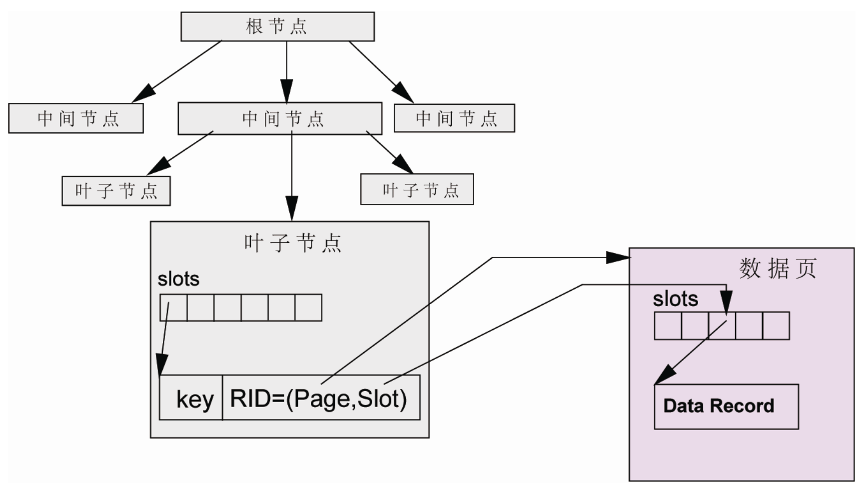


图 4-4　DB2 索引内部结构图

在 B+树结构中，还需要将前后叶子节点连接起来，以方便 RID 扫描。索引的叶子节点必须有向后的指针，也就是说支持正向扫描。创建索引 CREATE INDEX 语句中的 ALLOW REVERSE SCANS 子句将启用反向索引扫描，如果指定了 DISALLOW REVERSE SCANS，那么不能反向扫描索引。缺省情况下，索引允许按反向扫描。图 4-5 为双向索引的内部结构和指针情况。

索引的叶子是按照键值的顺序依次排列的，而双向索引同时包括两个方向的叶子节点的指针，指向左边的和右边的邻居叶子页。因此，双向索引可以用来从正向或反向（升序或降序）

扫描。

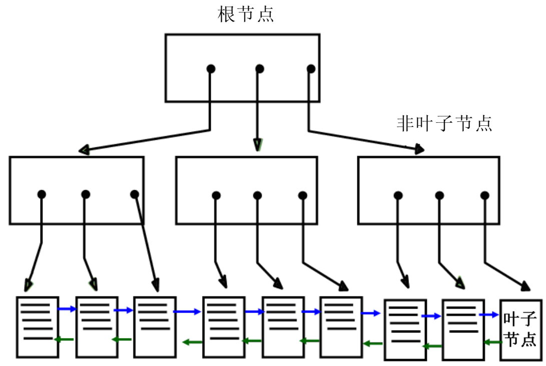


图 4-5　双向索引

聚集索引是一类特别的索引，它要求物理存储中的数据行顺序与索引中的 RID 顺序是对应的。在一个表中可以存在多个非聚集索引，但是只能存在一个聚集索引。如图 4-6 显示了聚集索引和非聚集索引的区别。对于聚集索引，DB2 尝试按照相应的键在索引页中的出现顺序把数据保存在数据页中，这样非常便于预取操作的执行，从而提高 I/O 性能。可以用 CREATE INDE 语句的 CLUSTER 子句来创建聚集索引，在小数据类型（比如 INTEGER）上或是查询中频繁用于范围搜索的字段上建立聚集索引。

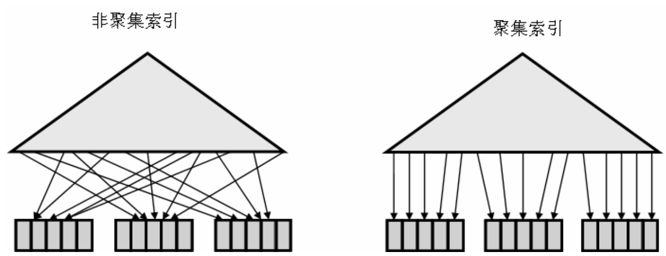


图 4-6　聚集索引和非聚集索引

从上面可以看出，索引的实质是以增大存储空间为代价来减少 I/O，它基于 B+树的结构具有下面 3 个特点：

- B+树结构：查找树型结构的时间复杂度为 O(LOG(n))，比表扫描的线型查找时间复杂度 O(n)，无疑快很多。
- 包含数据：索引上还包含索引字段的数据，在组合索引情况下，如果 SQL 语句只引用了索引上的字段，那么不需要访问表数据就可以得到查询结果。
- 键值有序：索引的键值是按大小顺序排列的，这不仅能加快查询速度，而且还可以为 SQL 语句中的 Order By 和 Group By 子句服务，从而避免排序。

4.5.2　为了性能设计索引

在什么条件下 DB2 会访问索引呢？这是由优化器根据 SQL 语句的 where 条件决定的。where 条件在 DB2 中被称为谓词，如图 4-7 所示，谓词主要有四类，分别是范围定界（Range-Delimiting）、索引控制（Index-Sargable）谓词、数据控制（Data-SARGable）谓词和保留（Residual）谓词，下面是它们的具体解释。

- Range-Delimiting 谓词用于限制索引扫描的范围，它们提供了索引扫描的起止位置。
- Index-Sargable 谓词不是用来限制搜索的范围，但由于谓词中的字段是索引的一部分，所以可以从索引中估计出来。Index-sargable 会减少访问基表的页的数目。但是并不能减少索引页的访问量。这两类谓词一般为“=”“>”“<”等谓词。
- Data-SARGable 谓词，索引管理器无法起作用，但是数据管理服务可以进行条件过滤。

通常，这些谓词需要访问基表。具有代表性的有“<>”、“LIKE”等。

- Residual 谓词的性能是最差的，所需的 I/O 操作非常大，已经超过了对基表的访问成本。Residual 谓词的例子包括关联子查询，使用 ANY、ALL、SOME 或 IN 子查询，或者读取 LONG VARCHAR 或是 LOB 数据等。

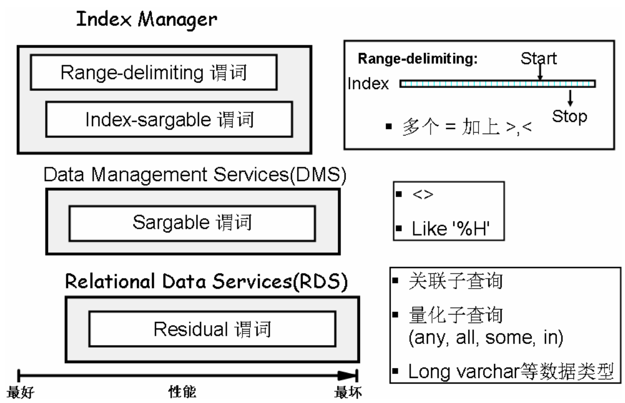


图 4-7　谓词分类与处理

注意：合理的索引也能为多表关联提供便利

如果是多表连接 SQL 语句，在内表（也称为被驱动表）的连接字段创建索引，也能在大部分情况下提高性能。读者可以参考本书第 9 章相关内容。

一个设计合理的索引，可以避免全表扫描、实现全索引访问、避免一定程度的排序，最终取得性能的提高。下面举例说明。

1. 避免全表扫描

当 DB2 执行一个 SQL 语句时，优化器会按照它的评估模型，对满足查询要求的各种方案的执行成本进行评估。最后，优化器会选择一个最优的访问计划，访问计划指定了各个操作的顺序。访问一个表中的数据有两种办法：

- 全表扫描：通过顺序扫描整个表，DB2 会依次访问表中的每一行。
- 索引扫描：首先访问表上的索引得到某一行的 RID，然后再从基表中访问该行。

可以从访问计划中判断是全表扫描还是索引扫描。例如，对于 SQL 语句“select acct_id, name from acct where balance >25000”，下面显示了它的访问计划，可以看到对表 ACCT 进行全表扫描，总成本为 11421.2：

```
Original Statement:
------------------
select acct_id, name
from acct
where balance > 25000

Access Plan:
-----------
    Total Cost:  11421.2
    Query Degree:    1
```

```
          Rows
         RETURN
         (   1)
          Cost
           I/O
           |
         5263.16
         TBSCAN
         (   2)
         11421.2
          2859
           |
         100000
    TABLE: DB2ADMIN
          ACCT
```

通常，全表扫描需要更多的 I/O 资源和缓冲区空间。为了避免全表扫描，需要对关键 SQL 语句进行分析，然后在相应的表上建立适当的索引。分析上面的这条 SQL 语句，可以在表 ACCT 的字段 BALANCE 上建立如下索引：

```
CREATE INDEX DB2ADMIN.IDX_ACCT_BALA  ON DB2ADMIN.ACCT(BALANCE ASC);
```

于是，DB2 在处理这条 SQL 语句时将用到新建的索引，相应的访问计划如下所示，可以看到索引扫描提升了查询性能，总成本从 11421.2 降为 1481.07：

```
Access Plan:
-----------
     Total Cost:       1481.07
     Query Degree:         1

           Rows
          RETURN
          (   1)
           Cost
            I/O
            |
          5263.16
          FETCH
          (   2)
          1481.07
            176
        /---+----\
    5263.16    100000
    IXSCAN   TABLE: DB2ADMIN
    (   3)        ACCT
    729.172        Q1
      52
      |
    100000
 INDEX: DB2ADMIN
  IDX_ACCT_BALA
       Q1
```

注意：索引扫描并不一定是最好的

如果使用索引扫描的成本更高时，优化器会选择全表扫描。通常下面几种情况很可能会使用全表扫描：

（1）表很小：直接访问表并不需要很多 IO 开销，比通过索引扫描然后再访问基表开销更低。

（2）索引聚集度很低：导致索引扫描后访问基表的 IO 增加，或者列表预取带来的排序代价过高。

（3）表的大部分行需要被访问：Where 条件中的谓词过滤度低，需要访问表的大部分数据，索引扫描没有好处。

2. 实现完全索引访问

如果 SQL 语句所需要的数据全部在某索引中，那么就没有必要访问相应的基表了，这就是通常所说的完全索引访问（Index-only Access）。

完全索引扫描的示例如下所示，组合索引 IDX_ACCT_GRP_ID 创建在字段 ACCT_GRP 和 ACCT_ID 上，而查询语句只用到了这两个字段，因此访问索引就能完成查询处理，不需要访问基表。

```
--创建索引--
CREATE UNIQUE INDEX IDX_ACCT_GRP_ID ON ACCT
    (ACCT_GRP ASC,
    ACCT_ID DESC);
--查询语句--
SELECT ACCT_ID, ACCT_GRP
FROM ACCT
WHERE ACCT_GRP = 20;
```

但是，如果 SQL 涉及的字段不全在索引中，那么就需要从基表中访问相应的行。例如，尽管有组合索引 IDX_ACCT_GRP_ID，仍然需要访问基表来获取字段 ACCT_NAME 和 ADDRESS 的数据。

```
SELECT ACCT_ID, ACCT_GRP, ACCT_NAME, ADDRESS
FROM ACCT
WHERE ACCT_GRP = 20;
```

如果某个关键 SQL 语句根据唯一索引（比如主键）来检索数据，而相应表的字段非常多，这时可以使用索引包含列（include）技术，即在建立唯一索引时，将 SQL 涉及的字段包含到索引的叶子节点中。这样，只需访问索引就可以完成处理，从而实现完全索引扫描。

使用索引包含列的例子如下所示，IDX_ACCT_UX 在字段 ACCT_GRP 和 ACCT_ID 上具有唯一性，同时在索引的叶子节点上也保存了字段 ACCT_NAME 和 ADDRESS 的数据。这样的话，下面的查询语句只需访问索引即可。

```
--创建唯一索引，并使用索引包含列技术
CREATE UNIQUE INDEX IDX_ACCT_UX ON ACCT
     (ACCT_GRP ASC,
```

```
          ACCT_ID DESC)
          INCLUDE (ACCT_NAME ASC, ADDRESS ASC);
    --查询语句--
    SELECT ACCT_ID, ACCT_GRP, ACCT_NAME, ADDRESS
    FROM ACCT
    WHERE ACCT_GRP = 20;
```

3. 避免排序

我们从上一节了解到，索引的键值是按大小顺序排列的，这样一来不仅可以提高查询性能，而且还可以为 SQL 语句中的 Order By 和 Group By 子句利用，从而避免排序要求。

比如，下面的 SQL 语句需要按照 ORDER BY ACCT_ID DESC 返回降序后的结果，而索引 IDX_ACCT_GRP_ID 是按照 ACCT_ID 降序创建的，所以可以从索引中直接获取降序后的结果，从而避免显式的排序：

```
    --创建索引--
    CREATE UNIQUE INDEX IDX_ACCT_GRP_ID ON ACCT
          (ACCT_GRP ASC,
          ACCT_ID DESC);
    SELECT ACCT_ID, ACCT_GRP, ACCT_NAME, ADDRESS
    FROM ACCT
    WHERE ACCT_GRP = 20
    ORDER BY ACCT_ID DESC;
```

在某航空公司数据库性能调优中，从监控中发现，一条关键 SQL 执行了 1765 万次，SQL 语句和访问计划如下所示，它需要按 FLIGHTDEPARTURETIME 降序返回结果，从下面的访问计划可以看出，做了显式的排序，总成本为 223.786：

```
    SELECT Q2."FLIGHTDETAILIDENTIFIER" AS "FLIGHTDETAILIDENTIFIER",
           Q2."FLIGHTIDENTIFIER" AS "FLIGHTIDENTIFIER", Q2."FLIGHTDEPARTURETIME"
           AS "FLIGHTDEPARTURETIME"
    FROM DB2ADMIN.FLIGHTDETAILHISTORY AS Q1, DB2ADMIN.FLIGHTDETAILHISTORY AS Q2
    WHERE (Q2."FLIGHTDETAILIDENTIFIER" = Q1."FLIGHTDETAILIDENTIFIER") AND
           (Q2."MATCHREQUIRED" = 'Y') AND (Q2."FLIGHTDEPARTURETIME" <=
           '2012-08-30-14.23.00.000000') AND ('2012-08-30-04.23.00.000000' <
           Q2."FLIGHTDEPARTURETIME") AND (Q2."FLIGHTNUMBER" = 'CDG4722      ')
           AND (Q2."RECORDTYPE" = 'RF') AND (Q2."RECORDSTATUS" = 'MA') AND
           (Q2."FLIGHTIDENTIFIER" <> 0)
    ORDER BY Q2."FLIGHTDEPARTURETIME" DESC
    Access Plan:
    -----------
        Total Cost:       223.786
        Query Degree:         1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
          2.85543e-005
```

```
            TBSCAN
            (   2)
            223.786
            197.49
              |
         2.85543e-005
            SORT
            (   3)
            223.786
            197.49
              |
         2.85543e-005
            FETCH
            (   4)
            223.785
            197.49
          /----+----\
     456.361     1.59032e+007
     RIDSCN     TABLE: DB2ADMIN
     (   5)   FLIGHTDETAILHISTORY
     23.8246
     3.11143
       |
     456.361
     SORT
     (   6)
     23.8243
     3.11143
       |
     456.361
     IXSCAN
     (   7)
     23.7255
     3.11143
       |
  1.59032e+007
 INDEX: DB2ADMIN
   X5ACPR37_32
```

于是检查索引 X5ACPR37_32 的定义，如下所示，字段 FLIGHTMOVEMENTTIME 没有起到任何作用。于是为该关键 SQL 新建索引 X6ACPR37_32，将字段 FLIGHTDEPARTURETIME 包含在新创建的索引中。从新的访问计划可以看出，显式的排序操作因为这个索引的出现而消除了，查询的总成本从 223.786 减少到了 22.7008，降低了 10 倍左右。

```
--原有索引的索引
CREATE INDEX "DB2ADMIN"."X5ACPR37_32" ON "DB2ADMIN"."FLIGHTDETAILHISTORY"
          ("FLIGHTNUMBER" ASC,
           "FLIGHTMOVEMENTTIME" ASC);
--修正后的新索引
CREATE INDEX "DB2ADMIN"."X6ACPR37_32" ON "DB2ADMIN"."FLIGHTDETAILHISTORY"
          ("FLIGHTNUMBER" ASC,
           "FLIGHTDEPARTURETIME" DESC)
```

```
--新索引后的访问计划
Access Plan:
-----------
    Total Cost:      22.7008
    Query Degree:        1

            Rows
           RETURN
           (   1)
            Cost
             I/O
             |
        2.85543e-005
           FETCH
           (   2)
           22.7008
           3.00003
        /-----+-----\
 2.86961e-005    1.59032e+007
   IXSCAN      TABLE: DB2ADMIN
   (   3)     FLIGHTDETAILHISTORY
   22.7004
      3
     |
 1.59032e+007
 INDEX: DB2ADMIN
  X6ACPR37_32
```

4.5.3 吃透组合索引

单字段索引好，还是组合索引好？这是很多设计人员经常喜欢问的问题，这个其实需要辩证去看。大多数情况下，组合索引比单字段索引用途广，但是其内部原理复杂。用好了组合索引，能极大提升系统性能，用不好反而对 DB2 来说是累赘。

组合索引之所以复杂，就在于字段排序不是随意指定的，而是有两个重要原则，一个是前缀性，一个是可选性。我见过很多设计人员，发现他们没有意识到应该优先设计组合索引，更没有充分理解组合索引的这两个重要原则。

下面我们重点探讨一下这两个主要原则：

第一个原则：前缀性（Prefixing），只有第一个字段作为过滤条件，索引才会启用。

不论单字段索引还是组合索引，它们在 DB2 内部都是有序的。如下数据所示，如果在部门、姓名和年龄这样的字段顺序建立索引（dept, name, age），那么先按照部门排序，再按照姓名排序，最后按照年龄排序。如果按照部门查询，索引肯定能用上，但如果只给出姓名进行查询，那么 DB2 就成了无头苍蝇，不会使用该索引了，就变成全表扫描了。

```
部门（dept）        姓名（name）        年龄（age）
信息技术部          张扬                48
信息技术部          张平                34
...
运维支撑部          王洋                26
```

```
运维支撑部          王东          35
...
```

第二个原则，选择性（selectivity）强的字段放在前面。

组合索引中如何排列字段顺序？DB2 建议按照选择性的强弱排序，选择性强的即字段值多的排在前面，从而可以提高查询性能。按照这个原则，上例中同一个部门的记录显然就太多了，具有相同姓名的却只有 1 个或几个，所以按照这个原则上面组合索引的字段顺序应为（name，age，dept）。

不过，这两个原则有时会冲突，就应该综合考虑。比如，要创建字段 name 和 dept 上的组合索引，假设 name 列的基数是 100000，dept 列的基数是 300。根据第二个原则，这个组合索引应当为（name，dept）。但是，如果实际的应用程序中，查询条件中单独使用 dept 字段的频率为 75%，单独使用 name 字段为 5%，同时使用 dept 和 name 字段为 20%。综合考虑，组合索引就应该是（dept，name）。

对很多读者来说，真正理解上面两个原则并不容易，下面就以不同的 where 条件为例来继续解读组合索引（name，age，dept）的用法。

（1）name = '张平' and age = 35 and dept = '运维支撑部'

这时，这三个字段上的谓词均可作为范围定界谓词。

（2）name = '张平' and age > 32 and dept = '运维支撑部'

首先，name = '张平'可以作为范围定界谓词，因为 name 是组合索引的第一个字段。

其次，age > 32 也可以作范围定界谓词，因为组合索引中 age 前面的字段只有 name，而 name 上正好有等于谓词。

最后，dept = '运维支撑部'不能作为范围定界谓词，因为组合索引中，dept 前面的字段 age 大于谓词。不过，dept = '运维支撑部'，可以作为 Index-Sargable 谓词，从而可以利用索引叶子节点上的数据进行计算。

小技巧：DB2 V10 新引入的 Index Jump Scan

当然，DB2 V10 中增强的 index jump scan 技术，通过枚举所有 age > 32 的 age 值，将 dept = '运维支撑部'包含为范围定界谓词，来构建新的索引查找区间。如果 age > 32 的 age 值数目较少，那么这个技术会很有用。

（3）name = '张平' and dept = '运维支撑部'

这时，name = '张平'可以作为范围定界谓词，但是 dept = '运维支撑部'只能作为 Index-Sargable 谓词，因为 where 条件中没有 age 字段的谓词。

请参考 4.5.2 节最后提到的 SQL 语句，理解为什么谓词'2012-07-01-00.00.00.000000' < Q1.ENDDATE 只能作为 Index-Sargable 谓词，而不能作为 Range-Delimiting 谓词。

注意：不等谓词的限制条件

如果一个字段上的条件为不等谓词（比如<、>、<=和>=），那么这个字段前面的所有字段必须都为等于（“=”）条件，这个字段才能作为范围定界（Range-Delimiting）谓词。

再回到本节开头提到的问题：单字段索引好，还是组合索引好？组合索引的字段顺序如何

安排？读者可以参考下面的建议：

（1）设计组合索引的时候，要重点考虑前缀性原则，只有索引中的第一个字段作为过滤条件，该组合索引才会启用。

（2）其次，要考虑选择性，选择性越强的字段，区分度就越好，就放在前面。

（3）分析 where 子句的过滤条件，如果过滤条件比较固定，则优先考虑创建组合索引。

（4）如果单个字段是主键或唯一性索引，或者是选择性非常强的字段，即使过滤条件比较固定，也不一定要建成组合索引，可以建成单字段索引，这样可以降低组合索引的开销。

（5）如果过滤条件涉及到的字段不固定，组合也比较灵活，则分别建立单字段索引。

（6）如果是多表连接 SQL 语句，可以在内表（也称为被驱动表）的连接字段与该表的其他约束条件字段建立组合索引。

4.5.4　创建索引

上面介绍了如何设计索引，那么如何在 DB2 中创建呢？这需要使用“CREATE INDEX”语句创建，它提供了多种选项，主要包括：

（1）UNIQUE：用来创建唯一索引，索引的键值不能重复。

（2）PCTFREE：每个叶子节点上预留的空闲空间，默认情况下 PCTFREE 为 10。PCTFREE 表示索引页中有多大的百分比是闲置的，在插入新的行和更新索引条目时，DB2 将利用这些空闲空间。如果这些空间被填满，那么 DB2 会进行索引页拆分操作。PCTFREE 参数的设置需要根据数据的读写进行区分，对于只读表，可以将 PCTFREE 设为 0；对于聚集索引或是存在大量插入操作的索引，建议 PCTFREE 值设的更大一些。

（3）ALLOW REVERSE SCANS：默认只支持对索引正向扫描，ALLOW REVERSE SCANS 指定该索引支持正向和反向扫描。

（4）INCLUDE：在唯一索引中可以使用 INCLUDE 子句将其他没有被索引的字段包含到索引叶子节点中，以实现完全索引访问。

（5）CLUSTER：用于创建聚集索引，聚集索引允许对数据页采用更线性的访问模式，允许更有效的预取，并且避免排序。

另外，DB2 除了支持关系数据上的索引，还能为 XML 数据和文本数据建立索引。

XML 索引机制和关系型数据库索引机制有类似之处，正如在关系查询中一样，索引对于保证 XQuery 和 SQL/XML 的性能是至关重要的。DB2 允许在 XML 字段上定义特定于路径的 XML 索引，这意味着可以使用它们来索引频繁在谓词和连接中使用的 XML 元素和属性。当应用程序向 DB2 提交关系查询或 XML 查询时，查询编译器将比较查询谓词和现有的索引定义，然后确定是否存在可用索引访问。

DB2 文本数据索引的建立，主要是为了对长数据类型，包括字符串类型、XML 类型和大对象类型（LOB）进行文本搜索。不同于关系型数据的模糊查询（例如 like 子句），它会对相应的文本进行分词处理，所以创建索引后，用户只需很短的时间就可以从索引得出需要搜索的关键字，扩展了数据库的功能。

4.5.5　索引的常见误区

索引是把双刃剑，一方面发挥其优势，即首先要定位关键 SQL 语句，分析这些 SQL 语句的特征，然后根据数据的实际分布特征建立合适的索引，并确保被有效使用。另一方面，不能

忽视索引维护的成本，索引太多对 INSERT、UPDATE 或 DELETE 操作频繁的数据库会有负面的影响。

接下来结合某航空公司的 SQL 优化实战案例，分享几个常见的索引使用误区，以飨读者。

1. 误区一：索引创建了就一定会启用

索引创建了并不代表一定会启用，例如有的索引是多余的，查询语句不可能用到，这个时候，可以用下面的语句查询数据库中从未使用或者很久没使用过的索引，其中 LASTUSED 字段表示该索引的最后一次使用时间：

```
select indschema,indname,tabname from syscat.indexes where LASTUSED='01/01/0001'
```

上面的方法只有在 DB2 V9.7 或更高版本才能使用，低版本的话，可以使用 db2pd 命令来获取，如下所示：

```
db2pd -db sample -tcbstats index
```

除了多余索引没有用到外，在一些特殊情况下会出现索引抑制现象。例如，在 flight_date 字段上创建了索引，但是对下面的 SQL 语句仍然无法启用，依然是全表扫描，这是因为索引树中是 flight_date 的值，不是 to_char(flight_date)的函数值。这样一来就出现了索引抑制，所以，不要轻易在字段前增加函数。

```
to_char(flight_date) between '2012-12-01' and '2012-12-28'
```

另外，尽量不要把索引字段嵌入到表达式中。例如下面的语句，即使在 flight_no 上创建索引，也不会启用索引扫描，原因是索引树上记录的是 flight_no，而不是 flight_no+6。

```
flight_no + 6 < 8340
```

那么，怎么解决呢？其实办法也很简单，把上面语句改成下面的代码，这样就能启用索引了。

```
flight_no  < 8340 - 6
```

2. 误区二：为了提高查询性能，同时创建组合索引和单字段索引

当一个表上索引较多而且需要优化的查询也较多时，就可能出现这种问题，根本原因还是对索引机制了解不够深刻。在某航空公司的 SQL 性能优化中，正是为了优化不同的 SQL，就出现了建重复索引的情况。如下所示，索引 CHARGE_DET_SERVICECODE 是多余的。

```
CREATE INDEX "DB2ADMIN"."CHARGEDE_INDEX5" ON "DB2ADMIN"."CHARGEDETERMINATION"
     ("SERVICECODE" ASC,
      "SERVICEPROVIDER" ASC,
      "ACTIVERECORD" ASC,
          "STARTDATE" ASC,
      "ENDDATE" ASC)
     CLUSTER ALLOW REVERSE SCANS;
CREATE INDEX "DB2ADMIN"."CHARGE_DET_SERVICECODE"
       ON "DB2ADMIN"."CHARGEDETERMINATION"
          ("SERVICECODE" ASC)
          DISALLOW REVERSE SCANS;
```

实际上，一个组合索引相当于多个索引，如索引(A, B, C)至少相当于(A)、(A, B)、(A, B, C)三个索引。这也是在 SQL 语句中把最多被访问的字段放到第一个位置的原因，这样的索引就能覆盖更多的查询。

3. 误区三：组合索引包含的字段越多越好

持续 2 天时间监控某航空公司的数据库，发现一条关键 SQL，它执行了 129837486 次，在两天的运行中，在这条 SQL 上总执行时间为十几个小时。当问起这条 SQL 是否用到了索引时，我们从客户那里得到的回答是肯定的。这条 SQL 语句和相关索引如下：

```
select UNIQUEIDENTIFIER, FILEDFLIGHTRULES, AIRCRAFTUSAGE
from DB2ADMIN.CHARGEDETERMINATION
where activeRecord='Y' and serviceCode='DISP' and serviceProvider='CATMBF'
     and STARTDATE < '2012-07-01 00:00:00' and ENDDATE>'2012-07-01 00:00:00'
order by UNIQUEIDENTIFIER asc

CREATE INDEX "DB2ADMIN"."CHARGEDE_INDEX5" ON "DB2ADMIN"."CHARGEDETERMINATION"
         ("SERVICECODE" ASC,
          "SERVICEPROVIDER" ASC,
          "ACTIVERECORD" ASC,
          "STARTDATE" ASC,
          "ENDDATE" ASC)
         CLUSTER ALLOW REVERSE SCANS;
```

从访问计划中，确实也发现这条 SQL 用到了该索引。请看访问计划中索引扫描中的谓词处理，Start Key Predicate 和 Stop Key Predicate 对应 4.5.1 节提到的 Range-Delimiting 谓词，而 IXSCAN 操作符下的 Sargable Predicate 对应 Index-Sargable。但是仔细分析，发现 ACTIVERECORD = 'Y'和 STARTDATE < '2012-07-01-00.00.00.000000'的过滤度（Filter Factor）太低，分别为 0.969627 和 0.99888。经过调查，发现几乎所有的相关查询都将这两个字段的谓词计算为 TRUE。那么，把这两个字段放在索引字段中完全没有必要，反而增加了索引 B+树的大小和层次，影响性能。

```
    5) IXSCAN: (Index Scan)
        ... ... 省略无关部分
        Predicates:
        ----------
        2) Start Key Predicate,
            Comparison Operator:      Equal (=)
            Subquery Input Required:  No
            Filter Factor:            0.0114613

            Predicate Text:
            --------------
            (Q1.SERVICEPROVIDER = 'CATMBF  ')


        2) Stop Key Predicate,
            Comparison Operator:      Equal (=)
            Subquery Input Required:  No
            Filter Factor:            0.0114613
```

```
        Predicate Text:
        --------------
        (Q1.SERVICEPROVIDER = 'CATMBF  ')


    3) Start Key Predicate,
        Comparison Operator:      Equal (=)
        Subquery Input Required:  No
        Filter Factor:            0.45788

        Predicate Text:
        --------------
        (Q1.SERVICECODE = 'DISP     ')


    3) Stop Key Predicate,
        Comparison Operator:      Equal (=)
        Subquery Input Required:  No
        Filter Factor:            0.45788

        Predicate Text:
        --------------
        (Q1.SERVICECODE = 'DISP     ')


    4) Start Key Predicate,
        Comparison Operator:      Equal (=)
        Subquery Input Required:  No
        Filter Factor:            0.969627

        Predicate Text:
        --------------
        (Q1.ACTIVERECORD = 'Y')

    4) Stop Key Predicate,
        Comparison Operator:      Equal (=)
        Subquery Input Required:  No
        Filter Factor:            0.969627

        Predicate Text:
        --------------
        (Q1.ACTIVERECORD = 'Y')

    5) Sargable Predicate,
        Comparison Operator:      Less Than (<)
        Subquery Input Required:  No
        Filter Factor:            0.802292

        Predicate Text:
        --------------
        ('2012-07-01-00.00.00.000000' < Q1.ENDDATE)
```

```
        6) Stop Key Predicate,
            Comparison Operator:      Less Than (<)
            Subquery Input Required:  No
            Filter Factor:            0.99888

            Predicate Text:
            --------------
            (Q1.STARTDATE < '2012-07-01-00.00.00.000000')
```

4. 误区四：函数索引很强大，可以多用

估计很多读者熟悉 Oracle 的函数索引，它计算函数的值，随后将最终结果存储在索引中。DB2 的实现机制不同于 Oracle，它是创建计算列用来存放函数产生的结果，并且为该计算列创建索引。

在下面的 DB2 例子中，我们分别在字段 first_name、mid_name 和 last_name 上调用 UPPER 函数并创建新的字段来存放 UPPER 函数计算后的结果。为了提升查询性能，又为新产生的计算列 upper_first_name、upper_mid_name 和 upper_last_name 特别创建了两个索引：ix_up_name_1 和 ix_up_name_2。语句如下所示：

```
SET INTEGRITY FOR NAME OFF;
ALTER TABLE name ADD COLUMN upper_first_name
GENERATED ALWAYS AS ( UPPER(first_name) );
ALTER TABLE name ADD COLUMN upper_mid_name
GENERATED ALWAYS AS ( UPPER(mid_name) );
ALTER TABLE name ADD COLUMN upper_last_name
GENERATED ALWAYS AS ( UPPER(last_name) );
SET INTEGRITY FOR name IMMEDIATE CHECKED FORCE GENERATED;
CREATE INDEX ix_up_name_1 ON name
(upper_first_name ASC,
upper_mid_name ASC) ;
CREATE INDEX ix_up_name_2 ON name
(upper_last_name ASC,
upper_mid_name ASC) ;
```

从这个例子可以看出，在 DB2 中函数索引的维护成本明显高于普通索引，而且需要有额外的计算列来存放，这将会消耗更多的存储空间。所以，建议尽量少使用函数索引，而是将函数进行转换。

5. 误区五：为热表创建的索引越多越好

一个热表有 48 个字段，插入、更新和删除操作非常频繁，但是其上竟然创建 9 个索引，从而导致事务吞吐量下降。索引就像是盐，它能调味，但太多了菜就咸了。尽管多建索引可以优化更多的查询，但在维护索引会有额外开销。所以，一般建议 OLTP 环境，每个表上的索引为 3 个左右；OLAP 环境，每个表上的索引为 5 个左右。

6. 误区六：索引不需要 DBA 过多维护

很多 DBA 认为，为 SQL 语句创建好索引就万事大吉了，这是错误的观点。实际上，索引和表一样，也需要一些手工维护工作。如下所示，当首次为表 WEIGHTSOURCE 创建索引后，

需要执行 runstats 操作，这样优化器才能从统计信息中感知索引的存在：

```
runstats on table DB2ADMIN.WEIGHTSOURCE with distribution on all columns and indexes all
```

特别是，当频繁对索引字段进行更新和删除操作，会对索引造成大量的碎片，从而影响索引的访问效率，并造成索引 I/O 的增加，所以需要定期 reorg 操作。如下所示，定期对表 WEIGHTSOURCE 执行 reorg 操作：

```
reorg indexes all for table DB2ADMIN.WEIGHTSOURCE
```

4.6　触发器设计一瞥

触发器（trigger）是定义在表上的数据库对象，在该表上做增删改操作时如果符合条件会触发它，并执行它所定义的逻辑。触发器可用于用户数据检查、定义和增强业务逻辑、检查数据完整性、记录审计信息和生成报警等。

关于 DB2 触发器的开发，请参考《从 Oracle 到 DB2 开发从容转身》相关内容，里面讲述了如何使用 SQL PL 语言开发触发器。

关于触发器的使用场景，不同的人有不同的看法。我认识一些客户，他们告诉我说，他们禁止使用触发器。是啊，触发器让数据库的逻辑变得混乱，就像一个不定时的炸弹，谁知道它何时会爆炸呢？因此，对于触发器的设计和使用，我想说四个字："慎用，少用！"但所谓"存在就是合理"，触发器在某些场合有它独到的用处。比如：

- 触发器可以在不改变应用程序的情况下，实现某些新的业务逻辑。
- 触发器可用来记录数据更改的历史信息，从而为审计所用。
- 触发器可以用于计算生成列的值。

触发器的类型包括：

- BEFORE 触发器：在插入、更新或者删除之前执行触发器定义的动作。
- AFTER 触发器：在更新、插入或删除操作之后执行的动作，用于更新反映表之间关系或一致性的其他表中的数据，以确保数据完整性。
- INSTEAD OF 触发器：触发器定义的逻辑会代替相应的操作，比如对不支持插入、更新和删除操作的视图执行这些操作，可以定义 INSTEAD OF 触发器转换相应的增删改操作。

下面我们举一个 AFTER 触发器的例子。当在库存表（stock）中更新了产品数量（qty）后，如果剩余的数量小于等于 5，触发器在另外一张 reorder 表中插入一条记录，提醒采购员该进货了。触发器定义为 FOR EACH ROW，表明对每一行更新的数据，如果满足 qty<=5，该触发器就会被激活一次。

```
CREATE TRIGGER reorder
  AFTER UPDATE OF qty ON stock
  REFERENCING NEW AS n
  FOR EACH ROW MODE DB2SQL
  when (n.qty <= 5)
  INSERT INTO reorder VALUES (n.itemno, current timestamp);
```

灵活地使用触发器可以方便实现某些业务逻辑，从而避免应用程序编码。但是使用触发器要尽量谨慎，特别在高并发的表上尽量避免创建过多的触发器，因为触发器本身有如下局限性：

（1）表上如果有过多的触发器会影响性能。这很好理解，因为进行增删改操作时，DB2 还要执行相应的触发器操作，触发器的业务逻辑可能还比较复杂，这对增删改的性能产生影响。

（2）触发器可能导致表的关系变得复杂：如果表上定义的触发器比较多，或是触发器之间的关系比较复杂，特别是有时会发生连续触发，这在设计时需要特别关注。

（3）过于隐秘：由于触发器的逻辑是隐藏在数据库层面，出现问题难以发现，难以调试。

在设计过程中，需要对触发器的优缺点进行评估，合理地权衡使用。例如可以对触发器与应用编程实现的优劣进行比较，如果确实需要采用触发器，那么它的实现逻辑最好不要过于复杂，否则调试起来很困难，也会对性能造成影响。

小技巧：可以使用 DataStudio 调试触发器

如果不可避免地需要使用复杂的触发器，可以使用 DataStudio 3.2 及以上版本来开发，它提供了强大的触发器开发调试功能。

4.7 小结

逻辑设计在数据库的整个生命周期中至关重要，如果前期的逻辑设计没有做好，一旦系统上线，后面调整的代价往往非常高。本章从逻辑设计的基本功说起，首先讲到了逻辑设计不仅要深刻理解业务需求，还应该一开始就考虑性能需求。然后介绍了如何做好基本表的设计，包括规范化、模式、类型、键、序列等各种数据对象；对存储过程开发中常用到的临时表，也在 4.3 节作了介绍，分别讲解了 DGTT 和 CGTT 的使用和它们之间的区别；对于时态数据的管理，DB2 V10 提供了新的特性，4.4 节通过各种实例深入讲述了系统时态表和应用时态表的设计和管理。在 4.5 节深入探讨了索引的内部结构，并通过实际调优案例分析了索引设计的各种误区和相应的最佳实践。

掌握基本表和索引的设计是逻辑设计的基本功。利用这些技术，可以为设计出一个高性能的数据库系统打下坚实的基础。

第5章 高级逻辑设计

毫无疑问，数据库的数据量早已迈入 TB 级的时代。我们在谈论数据量的时候，开始以 TB 为单位，例如某电信公司的业务系统每年产生的数据量是 20TB，这给数据库设计带来很大的挑战。如果只是依靠第 4 章提到的表和索引的基础设计技术就很难满足这种条件下的性能需求，必须使用更高级的设计技术才能解决问题。

本章将给大家讲解 DB2 的高级设计技术，包括 DPF（数据库分区功能）、表分区、MDC（多维集群）、MQT（物化查询表）和压缩技术等，这些高级技术的运用是一门追求综合平衡、充满辩证的哲学，也是经验和技术不断积累的艺术。只有充分了解 DB2 这些高级技术以及适用场景，才能设计出满足 TB 级数据管理需求的高质量解决方案。

5.1 如何处理 TB 级的数据

好的数据库设计和混乱的设计有天壤之别。在支持客户的经历中，我们遇到过各种各样的系统性能问题，大多数就是因为设计不当引起的。设计良好的系统，处理几十个 TB 的数据，仍然处之泰然，不慌不忙，仿佛蕴藏着无穷的能量；设计不好的系统，即使物理设备配置相当好，处理几十个 GB 的数据，却像蜗牛一样，怎么也跑不快，把人都急死了。

5.1.1 方法学指导

我们需要按照第 1 章讲到的自上而下方法学指导逻辑设计实践。在开始设计新系统时，就应该充分了解需求，站在性能优化的角度，掌握 DB2 提供的各种处理 TB 级数据的新技术，灵活运用各种技术的最佳实践，避免误区和地雷。

一句话，处理海量数据时，必须为性能而设计，必须站得高，看得远。

5.1.2 合理运用高级设计技术

我们遇到过很多存在性能问题的数据库系统，都是由于没有合理运用 DB2 提供的高级技术导致的。所以，除了方法学指导外，还要合理运用 DB2 提供的高级设计技术，主要包括分区特性、MQT 和深度压缩技术。下面先简单介绍一下，后续会有更深入的探讨。

注意：别被名字搞晕了

数据库分区：英文全称为 Database Partitioning Facility，简写为 DPF。
表分区：英文全称为 Table Partition、Range Partition 或者 Data Partition。
多维聚集：英文全称为 Multidimensional Clustering，简写为 MDC。
物化查询表：英文全称为 Materialized Query Table，简写为 MQT。

1. 分区特性

分区技术在原理上并不难，就是分而治之，也就是将一张大表按照一定规则划分为物理上的许多小表，而逻辑上仍然维持为一张大表。这样，分区对应用来说是透明的，即对应用无须做任何修改，就可从原来访问普通表改为访问分区表。

如表 5-1 所示，DB2 提供了 DPF、表分区和 MDC 三种分区技术，它们可以单独使用，也可以组合使用。

表 5-1　DPF、表分区和 MDC 的比较

分区种类	DPF	表分区	MDC
建表子句	DISTRIBUTE BY HASH	PARTITION BY RANGE	ORGANIZE BY DIMENSION
原理	将数据按照哈希算法分布到多个数据库分区上，以支持大规模数据的并行处理	将同一范围内的数据放在独立的数据分区中，支持快速地转入转出数据	将具有相同维度值的记录放在相同的物理块上

（续表）

分区种类	DPF	表分区	MDC
设计要点	选择合适的分布键，将记录均匀地分布在各数据库分区上，并提高连接中的表并置（Table Collocation）	基于数据转入转出的特点设计表分区，根据数据的热度将分区放在不同的表空间或者存储中，尽量创建分区索引提升性能	选择合适的维度列，否则会带来大量的存储空间浪费

注意：什么是表并置

在 DPF 多分区环境中，就针对一个特定查询，需要频繁访问多个表并进行连接操作，如果这些表中相关数据位于同一个分区中，那么连接成本就低，否则需要使用不同的策略在分区间复制数据。实现这种连接中的表处理方法，就是表并置。

2. MQT

MQT 是以查询结果为基础定义的一种物化视图。当处理一个复杂查询时，启用 MQT 查询重写，这样就能直接使用 MQT 中的数据，而不是使用基本表的数据进行重新计算，从而显著提升复杂查询的性能。报表、多维分析等应用可以通过 MQT 来提升性能。

3. 深度压缩

在关系数据库中，I/O 往往是数据库中开销最大的操作。对数据启用压缩，读取相同的数据只需更少的 I/O 次数，而且缓冲池中能容纳更多的数据，从而提高缓冲池的命中率；对日志启用压缩，那么可以减少日志的存储空间和日志归档所花费的时间；对备份启用压缩，可以减少备份的存储空间，从而减少备份和恢复所需要的时间。

注意：多留意 DB2 压缩技术的演进

DB2 提供了强大的数据压缩能力，不仅能对基本表的数据进行压缩，还能对索引、临时表、XML 数据等进行压缩，而且对各种类型的数据都有很好的压缩比。

从技术演进来看，IBM 在 DB2 压缩技术上一直保持持续的改进:

DB2 V9.1 引入了表级行压缩技术;

DB2 V9.5 中加入了自动字典创建（ADC）特性;

DB2 V9.7 提供了对 XML 数据和临时表的压缩支持;

DB2 V10.1 引入自适应（Adaptive）压缩技术，可以根据数据样本的不同灵活选择行压缩和页压缩。

5.2 DB2 DPF 设计

DB2 DPF 通常使用在数据仓库环境中，在第 10 章将会深入讲解数据仓库中的 DPF 特性，本小节仅介绍 DPF 环境下的表设计。

那么 DPF 采用什么样的架构呢？DPF 采用非共享（Share-Nothing）体系结构，包含多个独立的分区（也称为节点），每个分区都具有自己的资源，例如内存、CPU 和磁盘以及自己的数据、索引和事务日志。在 DPF 环境中，DB2 通过哈希函数计算分布键的散列值，随后根据散列值将表中数据分布到各个数据库分区中。因此，有时 DPF 也被称为 Hash 分区。

5.2.1 DB2 DPF 技术内幕

DPF 的分区可以分布在不同的物理机器上，同一台机器也可以划分多个逻辑分区，这种设计可以发挥 MPP 服务器的并行处理能力（关于什么是 MPP，什么是 SMP，请参考第 10 章）。例如，在数据仓库应用中，数据规模能达到数百个 TB，将分区部署到多台集群服务器上是一种很好的解决方案，这样复杂查询将受益于 DPF 的并行处理特性。

DB2 DPF 是基于并行性的体系架构，如图 5-1 所示。

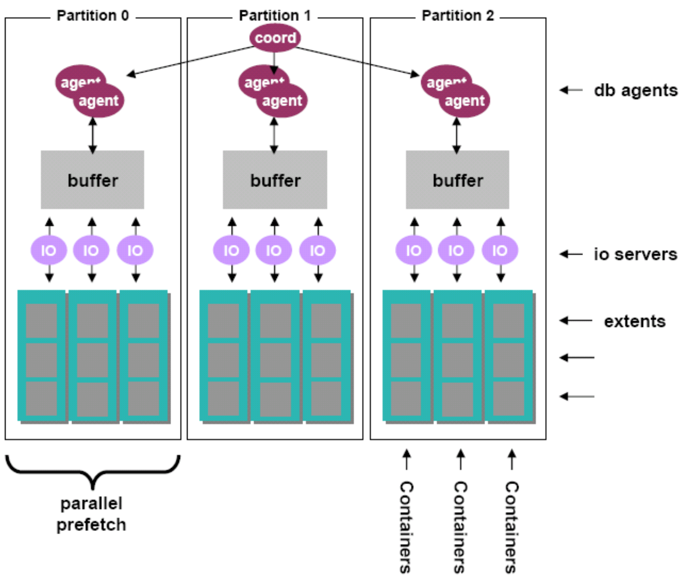


图 5-1 DPF 的并行架构

DPF 对应用来说是透明的。就如同向单分区 DB2 发出 SQL 请求一样，应用向 DPF 数据库发出请求，这时协调者节点（Coordinate Node，被连接的节点）接收到请求后，会将用户的请求分解成多个子任务交由不同节点并行处理，每个节点只负责处理自己的数据，最后协调者节点将各节点的执行结果经过汇总返回给应用。

那么如何定义表呢？在 CREATE TABLE 语句中使用 DISTRIBUTE BY 子句定义分布键。如下所示，SALES 表使用 S_ID 字段作为分布键，其中 TS_PD_N9 为分布在 9 个节点上的表空间。

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
    S_SOLD_DATE DATE NOT NULL,
    S_PRODUCT_SK INTEGER NOT NULL ,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
DISTRIBUTE BY HASH(S_ID)
IN TS_PD_N9;
```

DB2 将根据分布键的值将每行数据分布到各个节点上。具体过程如图 5-2 所示，DB2 根据分布键的值，利用 Hash 函数计算出一个介于 0~32767 的值，然后在 Hash 映射表中找到该行数据对应的分区。

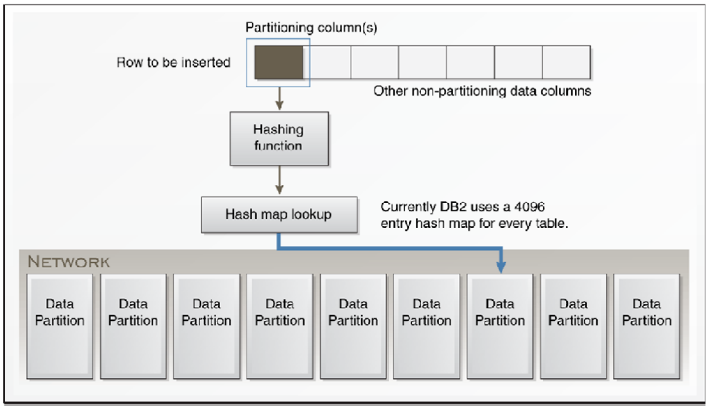


图 5-2　DPF 根据分布键分布数据

5.2.2　DB2 DPF 设计最佳实践

注意：DPF 设计的两大目标

第一，所有记录都能均匀的分布到各个分区上；

第二，选择合适的分布键实现表并置，将连接操作时分区之间的数据传输减少到最小。

首先，将大表放置在尽量多的分区上，将小表放置在尽量少的分区上。这样，大表是分区的，可以发挥并行处理的优势。如果大表需要和小表进行连接，那么小表也很容易被复制到多个分区上，不会占用过多网络带宽。

小技巧：DPF 设计中小表的处理

小表一般放置在单一的分区上，然后使用复制 MQT 技术将该表复制到多个分区上，从而实现连接并置。例如，产品表 product 是个大表，放置在 9 个分区上，表空间为 TS_PD_N9，而类别表 category 是个小表，放置在单一分区上。这时，可以使用下面的复制 MQT 语句将 category 表复制到 9 个分区上，从而实现 product 表和 category 表的并置连接。

```
CREATE TABLE m_category AS (
  SELECT * FROM category
  )
  DATA INITIALLY DEFERRED
  REFRESH IMMEDIATE
  IN TS_PD_N9
  REPLICATED;
REFRESH TABLE m_category;
```

其次，在 DPF 设计中，分布键的选择是重中之重，下面是选择分布键的一些最佳实践：

（1）选择基数（Cardinality）高的字段作为散列键。在这个原则下，主键和唯一键都是极好的选择。如果选择数据分布不均匀的字段或者基数低的字段作为散列键，则可能导致数据倾斜，也就是某些节点上数据多，而某些节点数据少，在处理查询时，可能需要等待数据最多的节点，才能由协调者节点汇总结果。

注意：什么是基数（Cardinality）

基数是 DB2 中非常重要的一个概念，特别是在分区设计和优化器中，基数是必须理解的一个概念。

表的基数：即表的行数。

列的基数：列上不同值的数目。例如，一共有 10000 种产品，那么订单表中列 PROD_SK 的基数最多为 10000，即使订单表有 1 亿条数据。在设计阶段，应该根据业务领域的知识或者样本数据来估计列的基数，如公司共有 20 个部门，一年最多 366 天，一年 12 个月，从而估计相应字段的基数。

（2）选择类型简单的字段，例如整型和定长字符类型，这能提高 Hash 算法的性能。

（3）避免选择更新频繁的字段作为分布键。

（4）选择查询中常用连接字段的子集作为分布键，以满足连接并置。

（5）将经常用于 GROUP BY 子句的字段包含在分布键中。

（6）将经常做连接的表放在同一数据库分区组中。

（7）使用经常要连接的字段作为分布键，以增加并置连接的数量。

5.3 表分区设计

表分区，也称为范围分区，或者叫数据分区，它是根据分区键的范围，将数据放在不同的分区中。每个分区包含该表的一部分记录，并且与其他记录分开存储。对于应用程序来说，表分区技术是透明的，应用看到的只是一张表。

表分区的每个分区都可以存放在相同或不同的表空间上，存放在不同的表空间上可以提升 I/O 吞吐量。图 5-3 展示了普通表和表分区在表空间的布局：Regular_T1 为普通表，只可以存放于一个表空间 Tablespace_2 中；表 RP_T1 有三个分区，都存储在同样的表空间 Tablespace_1 中；表 RP_T2 有两个分区，其中 P0 存储在 Tablespace_1，P1 存储在 Tablespace_3 中；RP_T3 有 4 个分区，其中 P0 存放在 Tablespace_1 中，P1、P2 和 P3

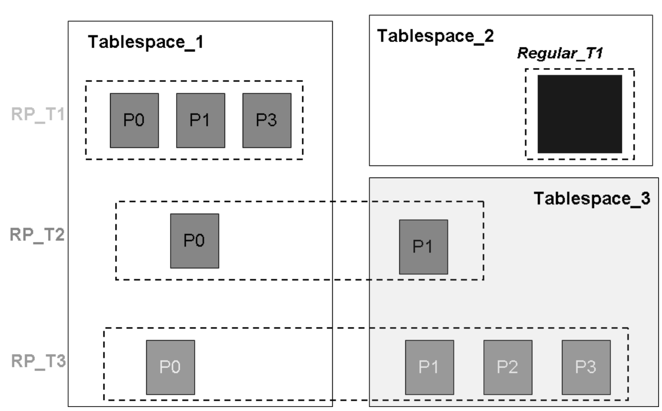


图 5-3　表分区和普通表的表空间布局

存放在 Tablespace_3 中。

5.3.1　表分区技术内幕

在很多项目的分区方案中，在指导思想上往往仅考虑部分目标，特别是性能需求，而对如何运用分区技术在批量数据处理方面考虑甚少。最佳分区设计方案应该是对性能需求和批量数据处理需求的全面考虑和综合平衡：

- 性能需求：如果没有分区，则会访问整个大表而不是相关分区，这样性能会受到影响。如果 SQL 语句在分区键上有谓词，DB2 能利用分区的范围定义排除不需要访问的分区，再结合分区索引，从而大大提高查询性能。
- 批量数据处理需求：在很多系统中，经常存在大批量数据迁移、数据转入转出等需求，通过分区技术与表空间设计相结合，可以在分区级完成上述工作，将大大提升批量数据的迁移性能。

先看如何创建表分区，根据 CREATE TABLE 语句中 PARTITION BY RANGE 子句的分区范围，将表的数据划分到多个数据分区中。如下例所示，按月份对销售表（SALES）进行范围分区，每个月份的数据放在一个独立的分区中。

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
S_SOLD_DATE DATE NOT NULL,
S_PRODUCT_SK INTEGER NOT NULL ,
S_CUSTOMER_SK INTEGER,
S_QUANTITY INTEGER,
    S_SALES_PRICE DECIMAL(7,2))
PARTITION BY RANGE(S_SOLD_DATE)
(PART Jan2012 STARTING '2012-01-01' ENDING '2012-01-31' IN TS1,
     PART Feb2012 STARTING '2012-02-01' ENDING '2012-02-29' IN TS2,
     PART Mar2012 STARTING '2012-03-01' ENDING '2012-03-31' IN TS3,
     PART Apr2012 STARTING '2012-04-01' ENDING '2012-04-30' IN TS4,
     PART May2012 STARTING '2012-05-01' ENDING '2012-05-31' IN TS5,
     PART Jun2012 STARTING '2012-06-01' ENDING '2012-06-30' IN TS6,
     PART Jul2012 STARTING '2012-07-01' ENDING '2012-07-31' IN TS7,
     PART Aug2012 STARTING '2012-08-01' ENDING '2012-08-31' IN TS8,
     PART Sep2012 STARTING '2012-09-01' ENDING '2012-09-30' IN TS9,
     PART Otc2012 STARTING '2012-10-01' ENDING '2012-10-31' IN TS10,
     PART Nov2012 STARTING '2012-11-01' ENDING '2012-11-30' IN TS11,
     PART Dec2012 STARTING '2012-12-01' ENDING '2012-12-31' IN TS12);
```

表 Sales 的分区结构如图 5-4 所示，对于应用程序来说，它们看到的只是一张表，如果所插入的数据行不满足任何一个分区范围的条件时，则会出现如下错误“SQL0327N : The row cannot be inserted because it is outside the bounds”。

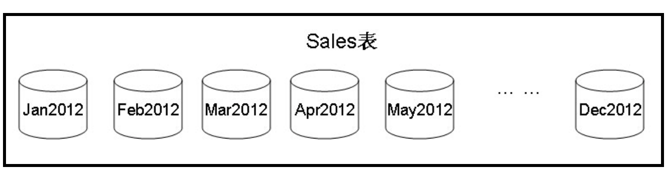


图 5-4　Sales 表的数据分区

创建分区表后，可以使用 DESCRIBE DATA PARTITIONS 命令查看各分区的定义，具体语

法为：

```
DESCRIBE DATA PARTITIONS FOR TABLE <rp_table_name> [SHOW DETAIL]
```

查看上例中表 SALES 的分区信息如下所示，分区的 ID、范围定义、分区名字和分区的表空间 ID 一目了然：

```
DESCRIBE DATA PARTITIONS FOR TABLE sales SHOW DETAIL;


PartitionId Inclusive (y/n)                Inclusive (y/n)
              Low Value                     High Value
----------- - ------------------------------ - ------------------------------
          0 Y '2012-01-01'                  Y '2012-01-31'
          1 Y '2012-02-01'                  Y '2012-02-29'
          2 Y '2012-03-01'                  Y '2012-03-31'
          3 Y '2012-04-01'                  Y '2012-04-30'
          4 Y '2012-05-01'                  Y '2012-05-31'
          5 Y '2012-06-01'                  Y '2012-06-30'
          6 Y '2012-07-01'                  Y '2012-07-31'
          7 Y '2012-08-01'                  Y '2012-08-31'
          8 Y '2012-09-01'                  Y '2012-09-30'
          9 Y '2012-10-01'                  Y '2012-10-31'
         10 Y '2012-11-01'                  Y '2012-11-30'
         11 Y '2012-12-01'                  Y '2012-12-31'

  12 record(s) selected.


PartitionId PartitionName                  TableSpId   PartObjId   IndexTblSpId
LongTblSpId AccessMode
                                                                          Status
----------- ------------------------------ ----------- ----------- ------------
----------- - ------
          0 JAN2012                                  7         267            7            7 F
          1 FEB2012                                  8           4            8            8 F
          2 MAR2012                                 20           4           20           20 F
          3 APR2012                                 21           4           21           21 F
          4 MAY2012                                 22           4           22           22 F
          5 JUN2012                                 23           4           23           23 F
          6 JUL2012                                 24           4           24           24 F
          7 AUG2012                                 25           4           25           25 F
          8 SEP2012                                 26           4           26           26 F
          9 OTC2012                                 27           4           27           27 F
         10 NOV2012                                 28           4           28           28 F
         11 DEC2012                                 29           4           29           29 F

  12 record(s) selected.
```

注意：分区表空间设计非常重要

分区设计中，可以指定每个分区存放于哪个表空间，即可以在分区级进行表空间设计。但是，很多项目在分区表空间设计上很随意，把所有分区都放到同一个表空间中。这导致 DBA 无法在表空间级进行与业务有关的数据管理工作，例如无法进行表空间级的数据迁移，无法进行表空间级的数据备份恢复。

上面创建表分区的例子似乎稍显复杂，它指定了分区名、各分区范围的起始值和结束值。事实上，DB2 提供了各种简洁的语法形式，方便用户使用。读者可以学习一下下面的几个例子。

例 1：创建表分区时可以不指定 ENDING 子句，而由 STARTING 定义分区范围。

```
CREATE TABLE SALES(S_SOLD_DATE DATE NOT NULL, S_CUSTOMER_SK INTEGER,...)
PARTITION BY RANGE(S_SOLD_DATE)
 (PART P1 STARTING '2012-01-01',
  PART P2 STARTING '2012-04-01',
  PART P3 STARTING '2012-07-01',
  PART P4 STARTING '2012-10-01' ENDING MAXVALUE
  )
```

例 2：创建表分区时用 ENDING 子句指定分区范围，分区名由系统生成。

```
CREATE TABLE SALES(S_SOLD_DATE DATE NOT NULL, S_CUSTOMER_SK INTEGER,...)
PARTITION BY RANGE(S_SOLD_DATE)
 (STARTING '2012-01-01' ENDING '2012-03-31',
  ENDING '2012-06-30',
  ENDING '2012-09-30',
  ENDING  MAXVALUE
  )
```

例 3：使用 EVERY 短语法创建表分区。

```
CREATE TABLE SALES(S_SOLD_DATE DATE NOT NULL, S_CUSTOMER_SK INTEGER,...)
PARTITION BY RANGE(S_SOLD_DATE)
   (STARTING '2012-01-01' ENDING '2012-12-31' EVERY 3 MONTHS)
```

例 4：指定多个字段作为分区键。

```
CREATE TABLE SALES(year INT, month INT, ...)
PARTITION BY RANGE(year, month)
(STARTING (2012, 1)  ENDING (2012, 3),
 STARTING (2012, 4)  ENDING (2012, 6),
 STARTING (2012, 7)  ENDING (2012, 9),
 STARTING (2012, 10) ENDING (2012, 12))
```

小技巧：如何找到分区表的分区名

由于在管理和维护表分区时，需要知道分区的名字，对于省略的分区名字或分区范围，可以用 DESCRIBE DATA PARTITIONS 命令或者查询系统编目表 SYSCAT. DATAPARTITIONS 得到。

注意：分区字段的数据类型

分区字段的数据类型必须是数值、字符或日期等基本类型，不能使用 LOBS 和 LONG VARCHARS。定义分区范围时有两个特殊值 MINVALUE 和 MAXVALUE，它们分别用来指定最小值和最大值。

最后，总结一下表分区的主要优势：首先，数据可以按范围分区并存放在不同的表空间中，这样可以方便数据的转入转出；其次，提高了 SQL 处理的性能，由于各个月份的数据分区存放，

如果用户需要访问某一特定月份的数据，可以直接对相应的分区访问即可，这样避免了访问不感兴趣的数据；再次，将数据放在不同的表空间，在进行数据访问时，可以进行并行 I/O，以提高访问效率；最后，使用表分区还有锁方面的好处是，当应用程序访问某一月份数据时，只需对相应的分区加锁，从而避免了像普通表一样对整个表加锁，这就提高了并发性。

5.3.2 全局索引 vs 分区索引

很多设计开发人员经常感到困惑：已经用上了分区表了，为什么性能还是没有提高啊？甚至有的更慢！原因很可能是没有合理设计分区索引，更深层次是没有搞明白 DB2 全局索引和分区索引的原理和适用场景。

在 DB2 V9.7 以前，表分区只支持全局索引（或者称为非分区索引），这样即使数据是分布在多个表空间上的，表分区的所有索引也只能存储在同一个表空间中。这约束了索引扫描的并行性，并对管理分区的转入转出带来不利影响。

DB2 V9.7 开始使用分区索引（局部索引）的组织方案，分区索引根据表的分区方案分散存储，每个索引分区都只引用对应数据分区中的表行。

1. 全局索引

全局索引作为整个表的单一对象来存储。创建全局索引时，必须使用 NOT PARTITIONED 子句。这可以在 CREATE TABLE 语句中使用 INDEX IN 子句来指定索引的默认存储表空间，也可以在 CREATE INDEX 语句中使用 IN 子句特别指定该索引的表空间。如果没有显示指定，全局索引默认创建在第一个数据分区所在的表空间。

如图 5-5 所示，在表 t1 上创建的两个索引 i1 和 i2 是全局索引，其中 i1 索引数据存放在表空间 tbsp4 中，i2 索引数据存放在表空间 tbsp5 中，它们都会索引所有分区的数据。

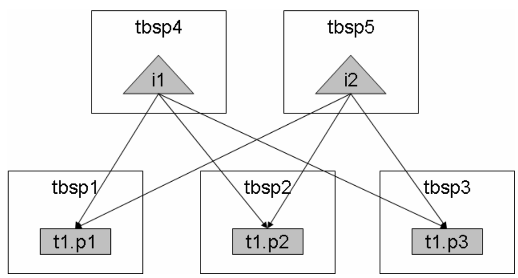


图 5-5 全局索引

```
CREATE TABLE t1(c1 INT, c2 INT)
   IN tbsp1, tbsp2, tbsp3
   INDEX IN tbsp4
   PARTITION BY RANGE(c1)
   (STARTING FROM (1) ENDING (100)EVERY (33))

CREATE INDEX i1 ON t1(c1) NOT PARTITIONED
CREATE INDEX i2 ON t1(c2) NOT PARTITIONED IN tbsp5
```

2. 分区索引

对于分区索引，每个表分区都有自己的索引分区，在 V9.7 以后，它是表分区创建索引时的默认类型。每个索引分区默认存放在相应数据分区所在的表空间，并可在 CREATE TABLE 语句定义各分区时用 INDEX IN 子句来特别指定该分区索引的表空间。

注意：创建分区索引的常见误区

很多设计人员在创建分区索引时希望和全局索引一样，用 CREATE INDEX 的 IN 子句指定分区索引的表空间，这是一个最常见误区。实际上，分区索引的表空间只能由 CREATE TABLE 定义各分区时的 INDEX IN 子句显示指定，否则默认就存放在数据所在的表空间。

如图 5-6 所示，索引 i1 为表 t1 的分区索引，每个数据分区的索引和相应的数据存放在一起，位于同一个表空间。索引 i2 为表 t1 上的全局索引，它位于独立的表空间 tbsp4 中。建表和建索引的 DDL 语句如下所示：

```
CREATE TABLE t1(c1 INT, c2 INT)
   IN tbsp1, tbsp2, tbsp3
   INDEX IN tbsp4
   PARTITION BY RANGE(c1)
   (STARTING FROM (1) ENDING (100)EVERY (33))

CREATE INDEX i1 ON t1(c1) PARTITIONED
CREATE INDEX i2 ON t1(c2) NON PARTITIONED
```

另外，图 5-6 还列出了表 t2 的索引分布情况。在创建表 t2 的语句中，用户指定了分区 1 和分区 2 的索引存放在表空间 tbsp5 中，分区 3 的分区索引则存放在和相应数据所在的表空间 tbsp3 中。建表和建索引的 DDL 语句如下所示：

```
CREATE TABLE t2(c1 INT, c2 INT)
 IN tbsp1, tbsp2, tbsp3
 INDEX IN tbsp3
 PARTITION BY RANGE(c1)
 (STARTING FROM (1) ENDING (33) INDEX IN tbsp5,
  STARTING FROM (34) ENDING (66) INDEX IN tbsp5,
  STARTING FROM (67) ENDING (100))
CREATE INDEX i3 ON t2(c1) PARTITIONED
```

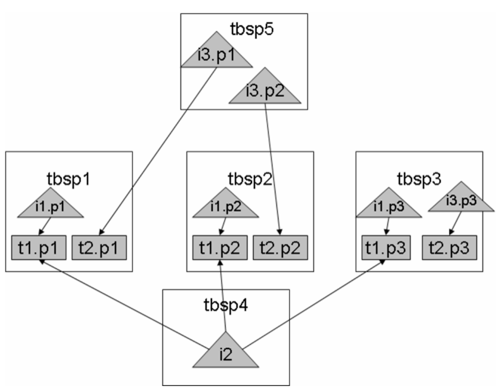


图 5-6　分区索引

3. 全局索引 vs 分区索引

首先，全局索引有自己的独特优势。如果一个查询需要读取多个分区的数据然后进行排序

或者聚集，那么在排序字段上建立全局索引可以避免额外的排序。例如，表 purchaseOrders 在 po_date 字段上按月进行分区，如果字段 po_customer 上的索引是分区索引，那么下面这个查询在索引扫描后将需要一次排序。

```
SELECT po_id, po_customer FROM purchaseOrders
    WHERE po_date > '2/15/2010' AND po_date < '4/15/2010'
    ORDER BY po_customer;
```

解决办法是在 po_customer 字段上建立全局索引，它能避免索引扫描后的排序。

与全局索引相比，分区索引的优势更明显：第一，在存储管理上，分区索引带来了更多精细的控制，并能提供索引的 I/O 并发访问；第二，结合分区排除技术，处理查询时只需访问相应数据分区的分区索引，减少了 I/O 操作，提高了索引扫描的性能；第三，分区索引能改善数据转入转出的性能，减少了这个过程中索引的维护开销。

所以，大家在具体设计工作中，要灵活运用，发挥不同索引的优势。

5.3.3 转入转出

通过表分区技术可以实现数据快速的转入转出，这给 DBA 维护大数据提供了极大的便利。例如，要删除某个大表中一个季度的数据，如果使用 delete 语句删除，那将需要漫长的过程，而通过表分区的转入转出可以让这个操作瞬间完成。

1. 转入操作

转入操作将一个现有表转换成分区表中的一个新数据分区，操作命令为 Attach，这个操作速度极快，对查询该表的应用程序造成的影响很小。

下面我们用例子说明，表 NewMonthSales 包含 2013 年 1 月份的销售数据，将其转入到表 SALES 中，过程如下。

第一步，利用 ALTER TABLE 语句的 ATTACH 命令，将表 NewMonthSales 加入到表 SALES 中，使其成为一个新的分区。执行 ATTACH 操作时需要获取表 SALES 上的 Z 锁，不过，ATTACH 操作不需要数据移动，也不需要维护索引，速度很快。执行 COMMIT 后，原来的数据分区可以被访问，但是新分区仍然不可见。

第二步，使用 SET INTEGRITY 命令完成剩余工作，包括验证数据、维护全局索引和局部索引、维护相应的 MQT 等。这些操作 DB2 会在后台执行，也就是说，当它运行时，原有的数据仍然可以正常使用。这一步结束并 COMMIT 后，转入操作就彻底完成了，新的数据分区就可用了。

```
--步骤 1:
ALTER TABLE  SALES
    ATTACH PARTITION  Jan2013
    STARTING '01/01/2013' ENDING '01/31/2013'
    FROM TABLE NewMonthSales;
COMMIT;

--步骤 2:
SET INTEGRITY FOR Sales IMMEDIATE CHECKED
COMMIT;
```

下面是转入数据最佳实践：

（1）尽量使用局部索引代替全局索引，从而加速数据的转入。

（2）数据转入前，在原表上创建合适的索引，使得与表分区的局部索引完全匹配。同时，在转入时，用 REQUIRE MATCHING INDEXES 子句强制索引匹配。

（3）在每一步完成后及时 COMMIT，释放相关的锁。

（4）保证原表上的数据满足各种约束，例如，其分区键的值应该满足分区定义的范围，从而减少 SET INTEGRITY 的时间。

2. 转出操作

转出操作的命令为 Detach，它提供了一种从分区表转出数据的极快方式。转出操作将现有的范围分区分离成单独的表，对原表来说该分区的数据立刻变得不可见，这对查询造成最小的干扰。这种方式适用于某个分区的数据当前不再需要，可以作为历史数据进行备份处理或者删除。

分区转出操作的语句示例如下：

```
ALTER TABLE SALES
   DETACH PARTITION Jan2012
   Into TABLE OldMonthSales
COMMIT;
```

转出操作的 DETACH 命令很简单，而且很快。这幕后的过程如下：

第一步，DETACH 语句将表 SALES 中分区 Jan2012 分离出去，使其成为一个独立的新表 OldMonthSales 表。执行该语句时，需要获取表 SALES 上的 Z 锁，不过不需要移动数据，也不会进行索引相关的数据维护，所以这个操作极快。

第二步，使用 COMMIT 语句提交后，SALES 表上的 Z 锁被释放，SALES 表的其他分区可以正常工作，而被分离的数据分区 Jan2012 变成不可见。在这个过程中，全局索引和分区索引并没有同步清理。

第三步，DB2 启动一个异步索引清除（AIC）线程，它在后台以较低优先级运行，用于清理和维护索引。与此同时，可以对分离的独立表 OldMonthSales 进行备份和删除等操作。

最后，总结一下转出数据最佳实践：

（1）在转出分区数据后，及时 COMMIT 以释放相关锁。

（2）可用 LIST UTILITIES SHOW DETAIL 来监控异步分区转出进程。

（3）如果需要转出多个分区，用 LIST UTILITIES SHOW DETAIL 命令监控前一个分区 DETACH 彻底完成后（包括索引清理和维护完成），再进行下一个分区的 DETACH 操作，避免资源竞争。

表分区的转入转出操作，最大的好处就是快，操作可以瞬间完成，对应用影响小。究其原因，就在于不引发数据移动，而数据和索引的维护在后台异步进行，不影响原有的数据。

5.3.4　分区排除技术

分区排除（Partition Elimination）就是 DB2 优化器能判断某些分区在处理查询时根本不需要被访问，从而提高查询性能。

如图 5-7 所示，现在要查询代码为 101920 的产品在二月中旬到三月中旬的销售总量，这时，DB2 将使用分区排除技术，判断该查询只需访问分区 Feb2012 和 Apr2012 两个分区。

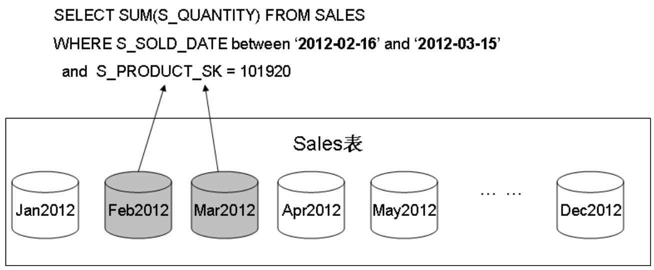


图 5-7 分区排除

在访问计划中，可以看到该查询分区排除的详细信息，如下所示：

```
Original Statement:
------------------
SELECT SUM(S_QUANTITY) FROM SALES
WHERE S_SOLD_DATE between '2012-02-16' and '2012-03-15'
  and  S_PRODUCT_SK = 101920

Optimized Statement:
-------------------
SELECT Q3.$C0
FROM
   (SELECT SUM(Q2.S_QUANTITY)
   FROM
      (SELECT Q1.S_QUANTITY
      FROM TPCDS.SALES AS Q1
      WHERE (Q1.S_PRODUCT_SK = 101920) AND (Q1.S_SOLD_DATE <= '2012-03-15') AND
             ('2012-02-16' <= Q1.S_SOLD_DATE)) AS Q2) AS Q3

Access Plan:
-----------
     Total Cost:       3082.42
     Query Degree:         1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
                1
             GRPBY
             (   2)
             3082.42
             2751.76
               |
             29266.6
             FETCH
             (   3)
             3078.39
             2751.76
          /----+-----\
       73166.5      7.31666e+06
```

```
       IXSCAN   DP-TABLE: TPCDS
       (   4)        SALES
       848.525         Q1
       721.12
         |
     7.31666e+06
 DP-INDEX: TPCDS
       S_PROD_IX
         Q1
... ... ... ...
     4) IXSCAN: (Index Scan)
          Cumulative Total Cost:        848.525
          Cumulative CPU Cost:      1.86138e+07
          Cumulative I/O Cost:      721.12
          Cumulative Re-Total Cost:     3.59868
          Cumulative Re-CPU Cost:   6.53036e+06
          Cumulative Re-I/O Cost:   0
          Cumulative First Row Cost:    15.1494
          Estimated Bufferpool Buffers:     723.12

          Arguments:
          ---------
          CUR_COMM: (Currently Committed)
              TRUE
          DPESTFLG: (Number of data partitions accessed are Estimated)
              FALSE
          DPLSTPRT: (List of data partitions accessed)
              1-2
          DPNUMPRT: (Number of data partitions accessed)
              2
          GLOBLOCK: (Global Lock intent)
              INTENT SHARE
          LCKAVOID: (Lock Avoidance)
              TRUE
          MAXPAGES: (Maximum pages for prefetch)
              719
          PREFETCH: (Type of Prefetch)
              SEQUENTIAL
          ROWLOCK : (Row Lock intent)
              SHARE (CS/RS)
          SCANDIR : (Scan Direction)
              FORWARD
          SKIP_INS: (Skip Inserted Rows)
              TRUE
          TABLOCK : (Table Lock intent)
              INTENT SHARE
          TBISOLVL: (Table access Isolation Level)
              CURSOR STABILITY

          Predicates:
          ----------
          2) Start Key Predicate,
              Comparison Operator:      Equal (=)
              Subquery Input Required:  No
              Filter Factor:            0.04

              Predicate Text:
```

```
          --------------
          (Q1.S_PRODUCT_SK = 101920)


      2) Stop Key Predicate,
          Comparison Operator:       Equal (=)
          Subquery Input Required:   No
          Filter Factor:             0.04

          Predicate Text:
          --------------
          (Q1.SS_CUSTOMER_SK = 1)

      DP Elim Predicates:
      ------------------
      Range 1)
          Stop  Predicate: (Q1.S_SOLD_DATE <= '2012-03-15')
          Start Predicate: ('2012-02-16' <= Q1.S_SOLD_DATE)
```

从上面访问计划中可以看到，TP-TABLE 表示 SALES 表为分区表，有 2 个分区被访问，即由 1-2 标识的这两个分区，用来做分区排除的谓词正是分区键 S_SOLD_DATE 上的查询条件。

5.3.5　分区维护

首先，分区键的选择决定了数据划分的方式。如果希望实现基于时间的转入转出，那么毫无疑问，使用时间（周、月或季）来进行分区是不错的选择。如果希望使用表分区作为一个性能解决方案，将表中的数据拆分存放为不同的物理对象（从而存储在不同表空间或磁盘上），那么选择国家、省份、地区、部门、子公司、产品分类等，都是合理的选择。

其次，控制分区的粒度和分区的个数。分区范围大小与通常数据转入转出的大小一致是理想的选择。如果分区范围太小，会导致分区数目过多，这将带来管理上的麻烦和空间浪费。

最后，尽量使用分区索引，因为它对转入转出性能带来了很大的好处，而且结合分区排除技术，在处理查询时分区索引比全局索引效率更高。

下面来看看表分区设计调整与维护的几个典型场景：

1. 将普通表转换成表分区

如何将一个普通表转换成分区表？是否可以直接用 ALTER TABLE 实现这种改变？遗憾的是，这种转换没有捷径可走，ALTER TABLE 帮不上忙。

这里提供一个方案，将普通表转换成分区表：

（1）创建一个分区表，字段定义和普通表完全一样，根据数据的热度将数据分区放在不同的表空间。

（2）将普通表的数据用 export、db2move 或者 ADMIN_MOVE_TABLE 存储过程导出。

（3）将数据用 LOAD 导入到新建的表分区中。

（4）创建合适的分区索引。

（5）用 DROP 语句删除普通表。

（6）用 RENAME 语句将分区表重命名，名字和普通表一样。

2. 增加新分区

如果分区表没有相应的分区定义，那么这时就需要增加新的分区，如下所示为 SALES 表新加一个分区 Jan2013。

```
ALTER TABLE Sales ADD PARTITION Jan2013
  STARTING ('2013-01-01') ENDING ('2012-01-31')
  IN Tbsp_data_2012M1 INDEX IN Tbsp_Idx_2012M1;
COMMIT;
```

比较：增加新分区 vs 转入分区

增加新分区只是增加空的分区定义，用于扩展分区以满足日益增长的业务需要。而转入分区则将现有的表转换成表分区的一个新数据分区，在增加新分区的同时也包含源表的数据。

小技巧：增加新分区技巧

增加新分区时，最好在一个维护窗口一次性增加多个分区。但是，也要避免增加太多分区，因为这样会影响表级别的统计信息，从而影响优化器选择访问计划。

3. 删除分区数据

出于某种原因需要将某分区的数据删除，但是需要保留分区的定义。用 DELETE 直接删除无疑太慢，而且会记录大量的日志，而 TRUNCATE 命令又不支持清除掉某分区的数据。那么怎么办呢？第一步，从 Sales 表转出分区 Jan2012 到表 Jan2012Sales；第二步，使用 TRUNCATE 命令清空 Jan2012Sales 表；第三步，重新从空表 Jan2012Sales 向 Sales 表转入新分区，这时分区 Jan2012 已经被清空了。

```
--STEP 1;
ALTER TABLE Sales DETACH PARTITION Jan2012 INTO Jan2012Sales;
COMMIT;
--STEP 2;
TRUNCATE TABLE Jan2012Sales IMMEDIATE;
--STEP 3;
ALTER TABLE Sales ATTACH PARTITION Jan2012
STARTING ('2012-01-01') ENDING ('2012-01-31')
FROM TABLE Jan2012Sales REQUIRE MATCHING INDEXES
```

5.3.6　V10 新特性：多温度存储

数据有冷暖，根据访问频率可以分为“冷（cold）”、“暖（warm）”和“热（hot）”。从 SATA 磁盘，到 SAS 盘，再到最新的 SSD 固态盘，I/O 速度越来越快，但是价格也越来越高。即使愿意花钱去买 SSD 盘，怎么会愿意将一些很少访问的数据放到 SSD 盘上呢？好钢用在刀刃上，SSD 盘当然应该用来存放关键应用的关键数据。

图 5-8　数据温度与存储层次

具体来说，数据温度与存储层次如图 5-8 所示，将访问频繁但数据量比较小的“热”数据存放在 SSD

盘，经常访问但数据量比较大的“暖”数据放在 FC/SAS 盘上，很少访问但数据量最大的“冷”数据存储在 SATA 盘上。这种存储层次需要利用分区表技术，根据数据热度将分区存放到不同的表空间，而表空间容器位于不同的磁盘上。不过，随着时间的推移，数据的热度会冷却，热数据会变成暖数据，暖数据会变成冷数据，这就需要一种机制在不同的存储层次间移动数据。

为了实现这一需求，基于数据的价值和访问模式的层次化存储管理方案应运而生，这就是 DB2 V10 的新特性“多温度存储架构”。这个新架构还引入了一个新概念“存储组（Storage Group）”，它介于表空间和磁盘之间，用来组织性能级别相同的磁盘。例如，将 SSD 盘组织成一个快速存储组，然后在上面建表空间用于存储热数据。

如图 5-9 所示，它结合了已有的表分区技术、自动存储表空间技术和新的存储组技术，对多温度的存储管理进行优化。横向来看，当前季度的数据 2012Q1 作为热数据放在 SSD 盘上，而一年之前的数据则作为冷数据放在 SATA 盘上。纵向可以看到 DB2 对存储的组织和管理，最底层是物理磁盘，物理磁盘被组织成存储组，“热”数据存储组为 SG_HOT，“暖”数据为 SG_WARM，“冷”数据为 SG_COLD，自动存储表空间创建在存储组上，分区表的各分区存放在不同的表空间中。

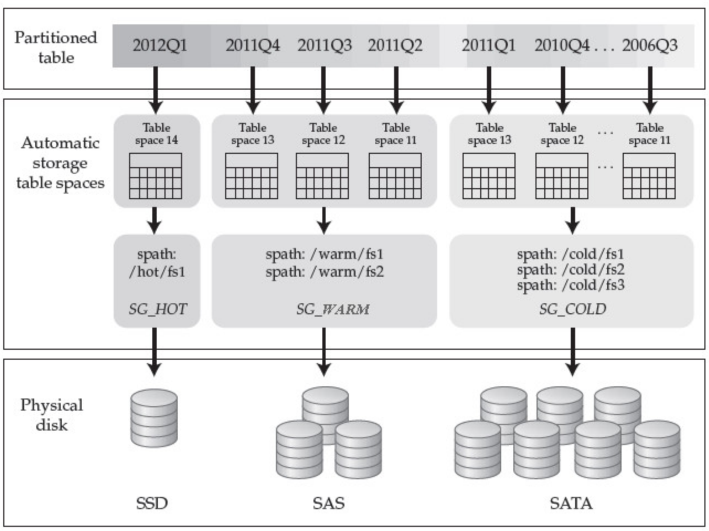


图 5-9　多数据存储解决方案

如何实现这个方案呢？首先来看如何创建存储组，如下所示，创建 3 个存储组将相同性能磁盘组织起来，其中 DATA TAG 属性用于标识数据访问的优先级，可以被工作负载管理器使用。

```
CREATE STOGROUP SG_HOT ON '/hot/fs1' DATA TAG 1;
CREATE STOGROUP SG_WARM ON '/warm/fs1', '/warm/fs2' DATA TAG 5;
CREATE STOGROUP SG_COLD ON '/cold/fs1', '/cold/fs2', '/cold/fs3' DATA TAG 9;
```

然后，根据存储组的特性和数据的热度为各分区创建表空间，如下所示：

```
CREATE TABLESPACE q1_2011 USING STOGROUP SG_COLD;
CREATE TABLESPACE q2_2011 USING STOGROUP SG_WARM;
CREATE TABLESPACE q3_2011 USING STOGROUP SG_WARM;
CREATE TABLESPACE q4_2011 USING STOGROUP SG_WARM;
CREATE TABLESPACE q1_2012 USING STOGROUP SG_HOT;
```

这时，就可以基于这些表空间创建分区表，将各分区放置在不同表空间中，如下所示：

```
CREATE TABLE SALES(S_SOLD_DATE DATE NOT NULL, S_CUSTOMER_SK INTEGER,...)
PARTITION BY RANGE(S_SOLD_DATE)
 (PART Q12011 STARTING '2011-01-01' IN q1_2011,
  PART Q22011 STARTING '2011-04-01' IN q2_2011,
  PART Q32011 STARTING '2011-07-01' IN q3_2011,
  PART Q42011 STARTING '2011-10-01' IN q4_2011,
  PART Q12012 STARTING '2012-01-01' ENDING '2012-03-31' IN q1_2012,
  );
```

新的季度到来时，就需要在存储组之间移动数据，例如将 Q22011 分区移到冷存储组 SG_COLD，而将 Q12012 分区移到暖分区组，这可以通过下面的命令简单实现：

```
ALTER TABLESPACE q2_2011 USING STOGROUP SG_COLD;
ALTER TABLESPACE q1_2012 USING STOGROUP SG_WARM;
```

为分区表添加新的分区，可将其存储在热存储组 SG_HOT 上，如下所示：

```
CREATE TABLESPACE q2_2012 USING STOGROUP SG_HOT;
ALTER TABLE Sales ADD PARTITION Q22012
  STARTING ('2012-04-01') ENDING ('2012-06-31') IN q2_2012;
```

最后，总结一下多温度存储技术带来的好处：

- 可以使用存储组来组织性能相同的磁盘，构建多温度存储层次，将热数据存储在快设备上，而将暖数据和冷数据存储在较慢的设备上。
- 便捷的数据管理。随着数据热度的冷却，需要在不同的存储层次间移动数据，可以通过简单的修改表空间的存储组来实现。
- 结合分区表技术和自动存储技术，多温度存储技术具有很强的实践意义。

5.4　MDC 表设计

MDC 把具有相同维度值的记录物理上放在一起，这种组织方式给数据处理带来诸多好处。例如，在查询的 where 子句中包含有维度字段的查询语句，能够很快地找到满足条件的数据块，从而显著提高查询的性能。

本节将具体探讨 MDC 的技术内幕和最佳实践。

5.4.1　MDC 技术内幕

先看一个 MDC 的具体例子，下面定义了一个 MDC 表 mdc_sales，按 year、nation 和 colour 这 3 个维度组织数据。

```
CREATE TABLE mdc_sales
(
   year INT NOT NULL,
   nation CHAR(10) NOT NULL,
```

```
    colour CHAR(6) NOT NULL,
    quantity INT NOT NULL,
    ... ...
) ORGANIZE BY DIMENSIONS (year, nation, colour);
```

这个表的数据组织方式如图 5-10 所示，具有相同维度值的数据放在一起，称之为一个单元（Cell），单元中存储空间的分配单位是块（block），一个块含有多个数据页。单元和块是理解 MDC 的关键概念，单元是 MDC 组织数据的逻辑单位，而块则是 MDC 分配空间的物理单位。

注意：什么是块（block）

块的另一个名字为扩展块（EXTENT），块的大小在创建表空间时指定，如下所示，表空间定义语句指定块的大小为 32 个页：

```
CREATE TABLESPACE ... EXTENTSIZE 32 PREFETCHSIZE 64...;
```

块是存储空间的分配单位，也就是说为 MDC 的单元增加存储空间时，一次增加一个块，而不是新增一个页。为了最大的 I/O 性能，块的大小（EXTENT SIZE）应该和 STRIPE SIZE 相匹配（相等或整数倍），请参考第 3 章 3.3.3 节。

注意：不要使用具有唯一性的字段作为维度列

这样会导致一个表所占用的空间急剧膨胀，原因是每个单元格的空间分配单位是块，即使这个单元格中只有一行数据，也会占据一个块的空间。

DB2 为 MDC 的每个维度自动地建立块索引（BLOCK INDEX），即以块为基本单位建立索引。普通表的行索引（RID INDEX）以数据行作为基本单位建立索引树。图 5-11 非常直观地体现了 MDC 的块索引和普通表的行索引的区别。

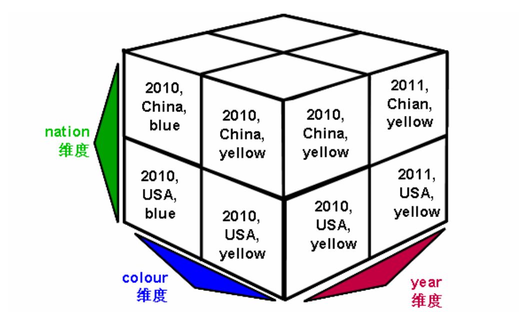


图 5-10 MDC 的数据组织

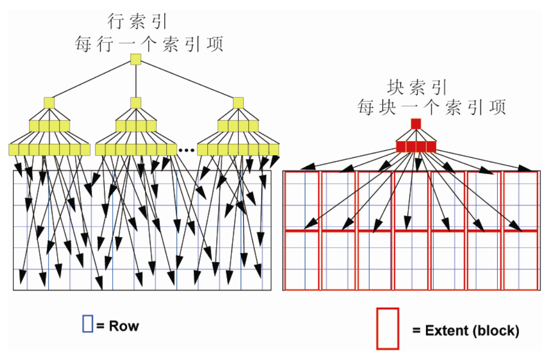


图 5-11 块索引和行索引

比较：块索引和普通索引的异同点

有三个重要区别：第一，块索引指向的是块，普通索引指向的是记录；第二，每个块包含了许多页的记录，所以块索引比行索引要小很多；第三，只在新块添加到单元，或者某些块需要从单元删除的时候，才需要被更新，因此，块索引需要的日志和维护方面的开销更小。

它们之间也有相同点：DB2 优化器会像处理普通索引一样对待块索引，块索引之间也可以

进行 AND 和 OR 操作，也可以和普通索引进行组合操作。

MDC 通常用于提高对大型表进行复杂查询的性能，特别适合那些在维度上做聚集计算的查询。例如：

```
SELECT SUM(quantity) FROM mdc_sales
WHERE year = 2011
GROUP BY nation, colour;
```

最后，总结一下 MDC 技术带来的各种好处：

首先，MDC 将维度值相同的数据组织在一起，这样数据更加有序从而提高 I/O 吞吐能力。

其次，聚集是自动维护的，在数据插入的过程中，MDC 会动态地管理和维护数据的聚集性，这样就不需要 REORG 来维护数据的聚集了。

最后，MDC 通过各维度上的块索引提高查询语句的性能，而且块索引比行索引占用的空间更小，效率更高。

5.4.2 MDC 表设计最佳实践

在 MDC 表中，每个单元都会根据需要来分配存储块，而块大小即为表空间的扩展块大小。如果 MDC 维度列选择不当，造成单元数目过多，那么即使某些单元仅存储一条记录，DB2 也会给这些单元分配独立的存储块，这样就会造成一定的空间浪费。由于需要读入一些空白页，这将导致性能下降。

关于 MDC 的设计，有下面一些注意事项。

（1）选择合适的维度列。

- 考虑性能关键的查询语句，关注 WHERE 条件中等于谓词或范围谓词所引用的列，以及 GROUP BY 和 ORDER BY 引用的列，可以作为候选的维度列。
- 考虑具有清晰区分度但基数不大的列，例如月份、星期、地区、省份或者部门都可以作为候选的维度列，避免选择主键或者基数很大的列。
- 事实表上的外键列也可以作为候选的 MDC 键。

（2）不要选择太多的维度。维度越多，单元个数就会以指数级速度相应地增长，从而产生大量的存储需求。如果表的存储空间过度膨胀（例如超过常规表的两倍大小），MDC 会因为稀疏块（部分填充）的 I/O 增加而使聚集失去优势。

（3）通过生成列（generated column）的方法来减少单元数量，避免空间浪费。例如，基于日期字段创建月份或星期生成列；或者用 Col/100 来代替 Col 作为维度列，这样，一个单元中的数据会明显增多，有利于空间利用。

（4）适当减少块大小，能相对减少单元中的空闲空间。块的大小需要在创建表空间时用 EXTENTSIZE 子句指定。

（5）使用 db2advis 来辅助 MDC 的设计。

最后，通过一个订单表 MDC_ORDER 的例子说明 MDC 键的选择。一个公司经营 20 类共计 10000 种产品，销售区域大约 30 个省，订单表含有销售日期字段 sold_date。

- 如果以（product，province，sold_date）作为 MDC 键，一年的数据总共有 1 亿个单元（10000×30×365=10950000），这极可能导致很多稀疏块或空页，造成空间浪费。
- 如果订单表的数据量更小些，可以用（category，province）作为 MDC 键，这时共有 600 个单元（20×30=600）。
- 如果以（category，province，year_mon）作为 MDC 键，一年的数据估计有 7200 个单元（20×30×12=7200），当订单表的数据行数达到几亿行，这个 MDC 键的选择是合适的。目前这个表的记录数为 3 亿左右，所以，最终选择了这 3 个列作为 MDC 键，其中 year_mon 为生成列。

```
CREATE TABLE MDC_ORDER
     (
      SOLD_DATE DATE,
      PROVINCE CHAR(25),
      PRODUCT INTEGER,
     CATEGORY INTEGER,
       ...
      YEAR_MON generated always as (INTEGER(SOLD_DATE)/100)
     ORGANIZE BY DIMENSIONS(CATEGORY,PROVINCE, YEAR_MON);
```

注意：

没有办法直接修改分区键，因为分区键的选择涉及到数据库表的设计问题，并且一旦建立就决定了数据如何分布，就只能使用表重建的方式来修改分区键。

5.4.3 案例："吃磁盘"的 MDC 表

在我接触到的客户中，不少客户使用 MDC 来提高性能。然而，设计 MDC 表时，维度键的选择至关重要，稍有不慎，不仅没能带来性能上的提高，还会发现 MDC 表在疯狂的"吃磁盘"。

在一次给某电信企业做数据仓库方面的技术服务时，就遇到过由于 MDC 键选择不当，导致占用空间过大。那是一个从 Oracle 迁移到 DB2 上的系统，在 Oracle 与 DB2 的对比测试中，发现表 TB_MK_DAY（只包含测试所用的部分数据）在 Oracle 中压缩后占用的空间只有 480M，而到了 DB2 里压缩后竟占用了 2.7GB 存储空间。整个表 TB_MK_DAY（包含全部数据）在 Oracle 上为 20G，而迁移到 DB2 的库竟然达到 120G。一时找不到原因，客户甚至为此事向 IBM 开出了问题报告（PMR）。而现场的工程师们一开始都怀疑是 DB2 的数据压缩特性有问题。

为什么迁移到 DB2 后，这个表会疯狂地"吃磁盘"呢？下面一一讲述。

首先，在数据迁移后的生产库上（包含 TB_MK_DAY 的全部数据），基于最新的统计信息，从 SYSCAT.TABLES 查询其相关数据，如下所示：

```
    select card,npages, fpages, compression,avgrowsize from syscat.tables where
tabname='TB_MK_DAY' and tabschema='MASA'

    CARD         NPAGES    FPAGES     COMPRESSION      AVGROWSIZE
    -------      --------- ---------  -------------   --------------------
    223336040    464218    3878040    R                58
```

针对上面的查询结果，用下面的公式计算，可以看出，此表占用的总空间为 FPAGES×

32KB （约 118.35GB）要远远大于表中实际行数所需要的空间 CARD× AVGROWSIZE（约 12.06GB），而浪费的空白页比率达到 88%，其中这个数据库的页大小为 32KB。

```
表占用的总空间： FPAGES * 32KB = 3878040 * 32KB = 118.35 GB
占用数据的页空间：NPAGES * 32KB = 464218 * 32KB = 14.17 GB
数据行的长度总和：CARD* AVGROWSIZE = 223336040 * 58 bytes = 12.06 GB
空白页数目：FPAGES - NPAGES = 3413822
浪费的空间比率：（FPAGES - NPAGES）/ FPAGES = 88%
```

经调查，得知 TB_MK_DAY 在 Oracle 上为哈希分区表，迁移到 DB2 上设计为 MDC 表，而 MDC 键为单字段 CEIL_ID。进一步调查，用如下语句分别在测试库和生产库上查询 CEIL_ID 的基数，发现测试库上 CEIL_ID 包含 6000 个不同的值，而生产库上约为 235000。

```
SELECT COUNT(DISTINCT CEIL_ID) FROM MASA.TB_MK_DAY;
```

最后，确认了存放 TB_MK_DAY 的表空间的块大小为 16，那么测试库浪费的空间约为：6000×16×32KB=2.9G，而现在表占用的空间为 2.7G，生产库浪费的空间更大，都符合上面的分析。

到这里，问题的原因已经清楚了，这是由于 MDC 键选择不当导致的。根据前面的最佳实践，在设计 MDC 表时，应该选择具有区分度但基数不大的列作为 MDC 键。

针对这个表，可以考虑用其他列做 MDC 键或取消对这个表做 MDC 分区。最终的解决办法是，选择了其他基数小的列做 MDC 键并重建这个表，经过在测试库上的验证，发现对创建后的表进行压缩，空间减为 380MB，这比 Oracle 的效果还要好！

注意：Oracle 与 DB2 的分区技术可不是一一对应的！

Oracle 的分区技术与 DB2 的分区技术并不一样，在做分区转换时，应该充分理解两者分区技术的特点和区别，千万不要像这个案例一样，将 Oracle 的哈希分区简单地转换成 DB2 的 MDC 分区。

5.5　分区技术对比：DB2 vs Oracle

Oracle 和 DB2 所支持的分区特性有相似之处，也有一定区别。为方便读者理解，表 5-2 以对比的方式介绍了 Oracle 和 DB2 这两种数据库的分区特性，请特别关注用法上的异同之处。当然，与 DB2 一样，这些分区特性在 Oracle 中既可以单独使用，也可以组合使用。

表 5-2　Oracle 和 DB2 分区特性比较

Oracle	DB2	Oracle 语法	DB2 语法
哈希分区	DPF	PARTITION BY HASH	DISTRIBUTE BY HASH
范围分区（Range Partitioning）	表分区	PARTITION BY RANGE	PARTITION BY RANGE

（续表）

Oracle	DB2	Oracle 语法	DB2 语法
列表分区（List Partitioning）	带生成列的表分区	PARTITION BY LIST	PARTITION BY RANGE
N/A	MDC	N/A	ORGANIZE BY DIMENSIONS

注意：系统分区和引用分区

Oracle 还支持系统分区、引用分区等特性，这在 DB2 中没有相对应的概念。

1. 哈希分区

DB2 的哈希分区（即 DPF 分区）与 Oracle 的哈希分区有着很大的不同。

注意：不同的哈希分区

DB2 的 DPF 采用 Share-Nothing 的体系结构，它扩展了存储和计算能力，包括 CPU、内存和存储等。Oracle 则是采用 Share-Disk（共享磁盘）的体系结构，利用哈希分区将数据分布到不同的表空间中，提高 I/O 并行性。

Oracle 使用如下所示的 PARTITION BY HASH 子句来建立哈希分区，其中哈希键为 S_ID，共创建了 4 个分区，分别存放于不同的表空间中。

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
    S_SOLD_DATE DATE NOT NULL,
    S_PRODUCT_SK INTEGER NOT NULL,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
PARTITION BY HASH(S_ID)
(partition p1 tablespace tbsp1,
 partition p2 tablespace tbsp2,
 partition p3 tablespace tbsp3,
 partition p4 tablespace tbsp4
)
```

那么 DB2 中是如何创建呢？如下所示，选择主键 S_ID 字段做分布键，根据 S_ID 的哈希值将数据分布到各节点上，其中 ts_data_4n 是跨 4 个节点的表空间。

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
    S_SOLD_DATE DATE NOT NULL,
    S_PRODUCT_SK INTEGER NOT NULL,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
  IN TABLESPACE ts_data_4n
  DISTRIBUTE BY HASH(S_ID)
```

2. 范围分区

表分区的数据组织形式是按照一个或多个字段将数据分布到多个表空间上，每个表空间独立存放表分区数据。

如下所示，在 Oracle 中为表 Sales 创建同样表分区的语句：

```
CREATE TABLE SALES(
     S_ID BIGINT NOT NULL,
     S_SOLD_DATE DATE NOT NULL,
     S_PRODUCT_SK INTEGER NOT NULL,
     S_CUSTOMER_SK INTEGER,
     S_REGION CHAR(5),
     S_QUANTITY INTEGER)
   PARTITION BY RANGE(S_SOLD_DATE)
    (PART Q1_2012 VALUES LESS THAN (TO_DATE('2012-03-31','YYYY-MM_DD')) TABLESPACE TBSP1,
    PART Q2_2012 VALUES LESS THAN (TO_DATE('2012-06-30','YYYY-MM_DD')) TABLESPACE TBSP2,
    PART Q3_2012 VALUES LESS THAN (TO_DATE('2012-09-30','YYYY-MM_DD')) TABLESPACE TBSP3,
    PART Q4_2012 VALUES LESS THAN (TO_DATE('2012-12-31','YYYY-MM_DD')) TABLESPACE TBSP4)
```

在 DB2 中为表 Sales 创建表分区如下所示，其中使用 S_SOLD_DATE 作为分区字段，每个季度作为一个分区：

```
CREATE TABLE SALES(
     S_ID BIGINT NOT NULL,
     S_SOLD_DATE DATE NOT NULL,
     S_PRODUCT_SK INTEGER NOT NULL,
     S_CUSTOMER_SK INTEGER,
     S_REGION CHAR(5),
     S_QUANTITY INTEGER)
   PARTITION BY RANGE(S_SOLD_DATE)
    (PART Q1_2012 STARTING '2012-01-01' ENDING '2012-03-31' IN TBSP1,
     PART Q2_2012 ENDING '2012-06-30' IN TBSP2,
     PART Q3_2012 ENDING '2012-09-30' IN TBSP3,
     PART Q4_2012 ENDING '2012-12-31' IN TBSP4)
```

3. 列表分区（List Partitioning）

列表分区是 Oracle 中的概念，其本质和范围分区一样，都是将数据存放到多个表分区，区别只是分区字段值是通过枚举出来的。如下所示，在 Oracle 数据库中将表 customer 创建为列表分区：

```
CREATE TABLE customer(
  cust_id int,
  cust_country char(2))
PARTITION BY LIST (cust_prov)
(PARTITION p_ea VALUES ('CN', 'JP', 'KR'),
 PARTITION p_na VALUES ('US', 'CA'),
 PARTITION p_eu VALUES ('EN', 'IT', 'DE', 'FR', 'RU'),
 PARTITION p_remain VALUES(DEFAULT))
```

DB2 可以通过表分区的方式来实现，具体来说，就是添加一个生成列（generated column），

并以此生成列作为分区字段。一种实现方法如下所示，在表 customer 中产生生成列 cust_ctr_gen，随后创建范围分区，从而达到和 Oracle 列表分区同样的效果：

```
CREATE TABLE customer(
cust_id INT,
cust_country CHAR(2),
cust_ctr_gen GENERATED ALWAYS AS
(CASE cust_country
    WHEN  'CN' THEN 1
    WHEN  'JP' THEN 1
    WHEN  'KR' THEN 1
    WHEN  'US' THEN 2
    WHEN  'CA' THEN 2
    WHEN  'EN' THEN 3
    WHEN  'IT' THEN 3
    WHEN  'DE' THEN 3
    WHEN  'FR' THEN 3
    WHEN  'RU' THEN 3
    ELSE 4
 END)
)
PARTITION BY RANGE (cust_ctr_gen)
(STARTING 1 ENDING 4 EVERY 1)
```

注意：分区字段的值要稳定才行

当然，这种方式一般要求分区字段的值域是相当稳定的，例如部门、地区等。笔者曾经遇到一个案例，客户要求将 Oracle 的列表分区转换成 DB2 中的表分区，但是列表分区字段是时间，那么这种转换方式就行不通。

4. 多维集群

DB2 中的 MDC 表按照多个维度来分块组织数据，例如产品、时间，区域等。如下所示，在 DB2 中创建了 MDC 表 Sales，其中以字段 S_PRODUCT_SK 和 S_REGION 作为维度列：

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
    S_SOLD_DATE DATE NOT NULL,
    S_PRODUCT_SK INTEGER NOT NULL,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
ORGANIZE BY DIMENSIONS (S_PRODUCT_SK, S_REGION)
```

在 Oracle 中，并没有与 MDC 等价的技术。请不要将聚簇表（Cluster Table）和索引组织表（Index Organization Table, IOT）与 MDC 技术混淆，它们是完全不同的技术。

Cluster Table 技术是存储一组表的方法。这些表具有某些相同的字段且更新频率不高，而经常在这些字段上做连接查询，于是就把这些不同表在这些连接字段上具有相同值的数据行存储到同一块上，从而提高性能。

索引组织表则是将数据行按表的主键顺序来组织数据，索引段就包括了所有数据行。

5. 组合方式（Combining Methods）

在实际应用中可以将上面的这些分区特性组合在一起使用，以满足多样化的应用需求。Oracle 可以哈希、范围、列表等组合，DB2 中可以 DPF、表分区分区和 MDC 组合。这样以应用 DPF 获得扩展性，并确保在逻辑分区之间均匀分布数据；表分区的转入转出能力可以方便数据管理，而分区排除技术能提高查询性能；MDC 可以用来数据聚集和提高查询性能。

如下所示，Oracle 中表 Sales 按照 S_SOLD_DATE 字段将每年的数据分布到不同的范围分区，对于每个范围分区又按照 S_ID 字段散列到各个子分区中：

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL PRIMARY KEY,
    S_SOLD_DATE DATE NOT NULL,
    S_PRODUCT_SK INTEGER NOT NULL,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
PARTITION BY RANGE(S_SOLD_DATE)
SUBPARTITION BY HASH(S_ID) SUBPARTITIONS 2
   (PART part1 VALUES LESS THAN (TO_DATE('2011-03-31','YYYY-MM_DD'))
     (SUBPARITION part11,
      SUBPARITION part12),
    PART PART2 VALUES LESS THAN (TO_DATE('2012-06-30','YYYY-MM_DD'))
     (SUBPARITION part21,
      SUBPARITION part22),
    PART PART3 VALUES LESS THAN (TO_DATE('2012-09-30','YYYY-MM_DD'))
     (SUBPARITION part31,
      SUBPARITION part32),
    PART PART4 VALUES LESS THAN (TO_DATE('2012-12-31','YYYY-MM_DD'))
     (SUBPARITION part41,
      SUBPARITION part42));
```

而相应的分区设计在 DB2 中显得更为简单，如下所示：

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL PRIMARY KEY,
S_SOLD_DATE DATE NOT NULL,
S_PRODUCT_SK INTEGER NOT NULL ,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
DISTRIBUTE BY HASH(S_ID)
PARTITION BY RANGE(S_SOLD_DATE)
( PART Q1_2012 STARTING MINVALUE ENDING '2012-03-31' IN q1_2012,
  PART Q2_2012 ENDING '2012-06-30' IN q2_2012,
  PART Q3_2012 ENDING '2012-09-30' IN q3_2012,
  PART Q3_2012 ENDING '2012-12-31' IN q3_2012);
```

在 DB2 中，DPF、表分区和 MDC 技术可以同时应用到一个表的设计中。如图 5-12 所示，从纵向看，表的数据分布在 3 个数据库分区上，而从横向看，表的数据按照月份进行范围分区，而 MDC 技术的使用又将相同特征的数据组织在一起。

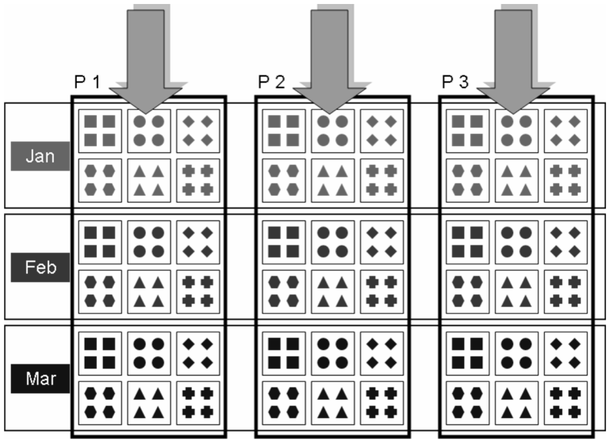


图 5-12 DPF、表分区和 MDC

结合 DPF、表分区和 MDC 三种技术的 SALES 表设计如下所示：SALES 表在 S_ID 上散列分布到各个数据库节点上；并在 S_SOLD_DATE 上按季度进行范围分区，存储在不同的表空间上；同时 SALE 表在 S_PRODUCT_SK 和 S_REGION 列上进行多维集群。

```
CREATE TABLE SALES(
    S_ID BIGINT NOT NULL,
S_SOLD_DATE DATE NOT NULL,
S_PRODUCT_SK INTEGER NOT NULL ,
    S_CUSTOMER_SK INTEGER,
    S_REGION CHAR(5),
    S_QUANTITY INTEGER)
DISTRIBUTE BY HASH(S_ID)
PARTITION BY RANGE(S_SOLD_DATE)
( PART Q1_2012 STARTING '2012-01-01' IN q1_2012,
  PART Q2_2012 STARTING '2012-04-01' IN q2_2012,
  PART Q3_2012 STARTING '2012-07-01' IN q3_2012,
  PART Q4_2012 STARTING '2012-10-01' ENDING '2012-12-31'  IN q4_2012)
ORGANIZE BY DIMENSIONS(S_PRODUCT_SK, S_REGION);
```

5.6 MQT 设计

MQT 是以查询结果为基础定义的一种物化视图。普通视图不同，MQT 是将查询结果保存到物理磁盘上。当一个复杂查询到达时，DB2 优化器确定整个查询或查询一部分是否可以用一个 MQT 来处理，于是用 MQT 重写查询，这样 DB2 直接使用 MQT 中的这些数据，而不是使用基本表的数据进行重新计算，这样可以显著提高复杂查询的性能。

MQT 最大的优势在于无须改变应用程序代码，就能提高复杂查询的响应时间，它在 OLAP 环境中取得了非常广泛的应用。

5.6.1 MQT 的能力

在笔者经历的一个银行应用调优中，月末报表的查询瓶颈在于一个 8 亿条记录的大表，在

该表上建立了 5 个 MQT 后，将其中一个报表的性能从 1 小时 15 分缩短到 10 分钟以内，而对该表上批量数据装载的性能影响不大，完全消除了最初对 MQT 会影响数据装载性能的担忧。

下面用一个例子来说明，下面的 SQL 语句查询每年的销售总量，在没有 MQT 的情况下，从访问计划上可以看出，两个表先做 JOIN 然后再做聚集 SUM 运算，优化器估计的开销为 2547.67。

```
Original Statement:
-------------------
SELECT YEAR(TX.pdate) tx_y, SUM(TI.amount) ti_amnt
FROM TRANS TX, TRANSITEM TI
WHERE TI.transid = TX.transid
GROUP BY YEAR(TX.pdate)

Optimized Statement:
--------------------
SELECT Q4.$C1 AS "TX_Y", Q4.$C0 AS "TI_AMNT"
FROM
   (SELECT SUM(Q3.$C1), Q3.$C0
   FROM
      (SELECT YEAR(Q2.PDATE), Q1.AMOUNT
      FROM STARS.TRANSITEM AS Q1, STARS.TRANS AS Q2
      WHERE (Q1.TRANSID = Q2.TRANSID)) AS Q3
   GROUP BY Q3.$C0 AS Q4)

Access Plan:
------------
       Total Cost:           2547.67
       Query Degree:          1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
               12
             GRPBY
             (   2)
             2547.44
               188
               |
               12
             GRPBY
             (   2)
             2547.44
               188
               |
               12
             TBSCAN
             (   3)
             2547.31
```

```
                 188
                  |
                 12
               SORT
               (   4)
               2547.13
                 188
                  |
                10000
               HSJOIN
               (   5)
               2003.45
                 188
         /------+-----\
      10000          5001
     TBSCAN         TBSCAN
     (   6)         (   7)
     1178.6         641.591
       116            72
        |              |
      10000          5001
 TABLE: STARS    TABLE: STARS
    TRANSITEM        TRANS
```

这时创建一个 MQT，该 MQT 统计交易表上每个月份的销售总量，并在 MQT 上创建索引，最后运行 runstats 命令更新统计信息。

```
CREATE TABLE SALES_BY_MONTH AS
( SELECT YEAR(TX.pdate) tx_year, MONTH(TX.pdate) tx_month,
         COUNT(*) item_cnt, SUM(TI.amount) total
  FROM TRANS TX, TRANSITEM TI
  WHERE TI.transid = TX.transid
  GROUP BY YEAR(TX.pdate), MONTH(TX.pdate))
DATA INITIALLY DEFERRED
REFRESH IMMEDIATE;

REFRESH TABLE SALES_BY_MONTH;

CREATE INDEX SALES_BY_MONTH_IX ON SALES_BY_MONTH(tx_year, tx_month);

RUNSTATS ON TABLE SALES_BY_MONTH
  WITH DISTRIBUTION AND DETAILED INDEXES ALL;
```

下面的访问计划是创建 MQT 之后的结果，可以看到：DB2 优化器用 MQT 重写了原来的 SQL 查询语句，并直接从 MQT 读取数据，这样使用 MQT 的访问计划没有了 JOIN 操作，整个查询的开销变为 34.6916，远远小于原来的开销。这就是 MQT 的能力！

```
Original Statement:
-------------------
SELECT YEAR(TX.pdate) tx_y, SUM(TI.amount) ti_amnt
FROM TRANS TX, TRANSITEM TI
WHERE TI.transid = TX.transid
GROUP BY YEAR(TX.pdate)
```

```
Optimized Statement:
--------------------
SELECT Q3.$C0 AS "TX_Y", Q3.$C1 AS "TI_AMNT"
FROM
   (SELECT Q2.$C0, SUM(Q2.$C1)
   FROM
      (SELECT Q1.TX_Y, Q1.TI_AMNT
      FROM STARS.SALES_BY_MONTH AS Q1)) AS Q2
   GROUP BY Q2.$C0) AS Q3

Access Plan:
-----------
     Total Cost:       34.6916
     Query Degree:         1

      Rows
     RETURN
     (   1)
      Cost
       I/O
       |
       7.1
     GRPBY
     (   2)
     34.5474
        1
       |
       7.1
     TBSCAN
     (   3)
     34.4664
        1
       |
       19
     SORT
     (   4)
     34.334
        1
       |
       71
     TBSCAN
     (   5)
     31.8631
        1
       |
       71
 TABLE: STARS
   SALES_BY_MONTH
       Q1
```

5.6.2　创建 MQT 必须掌握的要点

MQT 通过 CREATE TABLE 语句来创建，MQT 中引用的表称为基本表。图 5-13 展示了创建 MQT 的详细语法，其中标有下划线的选项是默认选项，即 MQT 默认创建为 MAINTAINED BY

SYSTEM，REFRESH DEFERRED 和 ENABLE QUERY OPTIMIZATION，下面具体谈谈这些选项的含义。

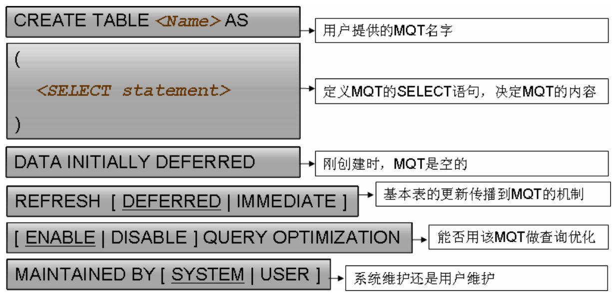


图 5-13　MQT 语法

- DATA INITIALLY DEFERRED：这是必需的选项，表示在执行 CREATE TABLE 语句时，并不将数据插入到表中。MQT 被创建好后，处于检查暂挂（check pending）状态，在对它执行 SET INTEGRITY 语句之前，不能查询它。执行 SET INTEGRITY 时，DB2 根据 MQT 的查询定义对数据进行检查，并将查询结果填充到 MQT。
- MAINTAINED BY [SYSTEM | USER]：包括系统维护和用户维护两种。系统维护的 MQT，不允许用户对 MQT 做 IUD（Insert、Update 或 Delete）操作，基本表的数据更改通过某种机制自动刷新到 MQT 中，大多数应用的 MQT 都创建为系统维护；用户维护的 MQT 则是由用户通过 IUD 或者 LOAD 操作装入数据，可以用在时效要求较低的应用中。
- REFRESH [DEFERRED | IMMEDIATE]：包括延时刷新和即时刷新两种。延时刷新需要通过命令 REFRESH TABLE 语句来刷新，即时刷新表示基本表的 IUD 操作带来的数据更新立即被刷新到 MQT。系统维护 MQT 采用延时刷新对基本表的影响很小，对于用户维护 MQT，只能采用即时刷新。
- [ENABLE | DISABLE] QUERY OPTIMIZATION：是否允许 DB2 优化器使用该 MQT 来优化复杂的查询。一般来说，使用 MQT 进行查询优化是创建 MQT 的动机，也有一些场合应用程序直接查询 MQT 中的数据。

5.6.3　深入讨论 MQT 的刷新机制

在现实使用中，用的最广泛的是系统维护的 MQT，它在提高复杂查询性能的同时，会将基本表的数据更新传播到 MQT，所以对基本表有一定影响。很多读者非常关心，这种影响到底有多大？其实这和它的刷新时机有关。

- 即时刷新：增量刷新方式，用于对时效性要求高的场景，但是对高负载的 OLTP 应用会有一定性能影响。
- 延时刷新：全量刷新方式，导致 MQT 的数据与基本表的数据不完全同步，一般用于时效性较低的应用，例如每月报表或者季度报表等。
- 混合型：结合上面两种模式，将基本表的更新即时传播到一个过渡表（staging table）中，然后在适当的时候用 REFRESH 命令将中间表的数据更新到 MQT。创建 MQT 的刷新选项为 REFRESH DEFERRED。

下面通过分析一个 INSERT 语句的访问计划来研究即时刷新 MQT 的数据传播机制。从图 5-14 的访问计划中可以看到，在 INSERT 操作之上，有一套复杂的机制将这条新插入的数据刷新到 MQT。

```
CREATE TABLE SALES_BY_TRANSID AS
( SELECT transid, COUNT(*) as item_count, SUM(amount) as total
   FROM TRANSITEM
   GROUP BY transid )
DATA INITIALLY DEFERRED
REFRESH IMMEDIATE;

REFRESH TABLE SALES_BY_TRANSID;

CREATE INDEX SALES_BY_TID_IX ON SALES_BY_TRANSID(transid);

RUNSTATS ON TABLE SALES_BY_TRANSID
  WITH DISTRIBUTION AND DETAILED INDEXES ALL;

EXPLAIN PLAN FOR
  INSERT INTO TRANSITEM VALUES('1004', '2808', 200.00, '2');
```

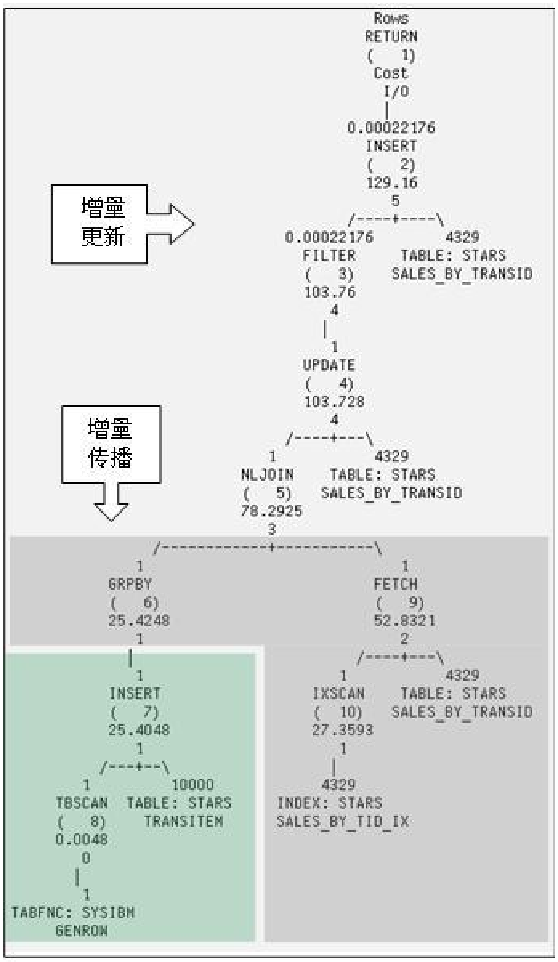


图 5-14　MQT 即时刷新的内部机制

来看一下利用过渡表的MQT刷新机制。为MQT创建过渡表和利用过渡表刷新MQT的SQL语句如下所示：

```
CREATE TABLE SALES_BY_TRANSID AS
( SELECT transid, COUNT(*) as item_count, SUM(amount) as total
   FROM TRANSITEM
   GROUP BY transid )
DATA INITIALLY DEFERRED
REFRESH DEFERRED;

CREATE TABLE STG_SALES_BY_TRANSID FOR SALES_BY_TRANSID PROPAGATE IMMEDIATE;

set integrity for SALES_BY_TRANSID materialized query immediate unchecked;
set integrity for STG_SALES_BY_TRANSID staging immediate unchecked;

REFRESH TABLE SALES_BY_TRANSID;

CREATE INDEX SALES_BY_TID_IX ON SALES_BY_TRANSID(transid);

RUNSTATS ON TABLE SALES_BY_TRANSID
  WITH DISTRIBUTION AND DETAILED INDEXES ALL;

INSERT INTO TRANSITEM SELECT ...;
REFRESH TABLE SALES_BY_TRANSID INCREMENTAL;
```

那么这种刷新的内部机制如何呢？如图5-15所示，它采用一种折中的方式，对基本表的更新只是传播到过渡表中，此时不会对 MQT 表做任何更新；而过渡表的数据则通过 REFRESH 命令（例如在运维窗口）刷新到MQT中。由于过渡表的数据与MQT相比一般较少，将基本表的更新即时传播到过渡表速度很快，因此，这种方式对基本表上插入、更新和删除操作影响也较小。

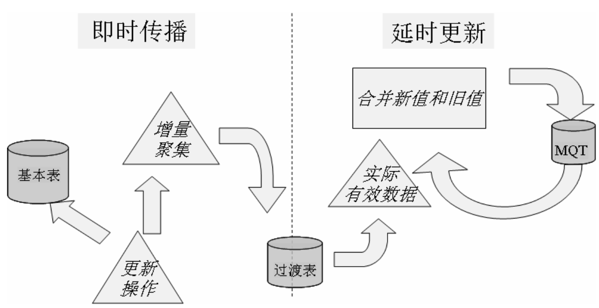


图 5-15　利用过渡表的MQT刷新机制

实际上，MQT对基本表的影响较小，一般为5%~10%的开销，但是对报表分析中复杂查询性能的提高往往是几倍甚至几十倍，所以总体来看，是用小的投入换来大的回报。

5.6.4　MQT 设计最佳实践

MQT 一般用于 OLAP 应用的复杂查询，如报表和分析应用。下面总结一下设计 MQT 的最佳实践：

（1）对含有 GROUP BY 子句和聚集操作（SUM、COUNT 和 AVG 等）的关键查询创建

MQT。因为聚集操作本身的成本很高，涉及大量的 I/O 操作，如果直接访问 MQT，性能会得到很大的提高。

（2）创建具有通用性的 MQT，并避免在 WHERE 条件中设置太多的局部谓词，使它能为更多的用户查询利用。例如，针对下面用于分析销售业绩的各种报表查询，创建的 MQT 在 deptment、product 和 year_mon 三个字段上进行分组聚集，于是下面的这些查询均可用到这个 MQT。

```
查询 1：
SELECT SUM() ... FROM SALES WHERE deptment = 'A'
 GROUP BY product, year_mon;
查询 2：
SELECT SUM() ... FROM SALES WHERE product = 'IPAD'
 GROUP BY deptment, year_mon;
查询 3：
SELECT SUM() ... FROM SALES  WHERE year_mon = 201110
   GROUP BY deptment, product;
查询 4：
SELECT SUM() ... FROM SALES
   GROUP BY product, year_mon;
... ...
创建的 MQT 如下：
CREATE TABLE M_SALES as (
   SELECT SUM() ... FROM SALES
   GROUP BY deptment, product, year_mon
) ... ...;
```

（3）针对大数据量的 MQT 表，可以为其创建合适的索引，从而进一步提高复杂查询的性能。

（4）MQT 并不是越多越好，它不仅会占用存储空间，而且会增加编译时间。因此需要有针对性地创建 MQT，保证其能够真正带来性能上的提升。

（5）MQT 会在一定程度上影响基本表的插入、更新和删除操作的性能，因此要权衡利弊，选择合适的刷新机制。

（6）可以利用 db2advis 工具辅助 MQT 表的设计。

很多读者可能遇到过这样的情况，成功创建了 MQT，但是并未被预期使用，那么该如何检查和诊断呢？

（1）确保 CURRENT REFRESH AGE 注册变量设置为 ANY，这样 DB2 才会考虑用 MQT 来优化查询，如下所示：

```
db2 SET CURRENT REFRESH AGE ANY
```

（2）确保对 MQT 启用了优化选项，即在创建或者更改 MQT 的语句指定了 ENABLE QUERY OPTIMIZATION 选项，这样优化器才会考虑此 MQT。

（3）分析用户查询并与 MQT 的定义比较，确保查询能用该 MQT 来优化。

（4）运行 RUNSTATS 收集 MQT 和基本表的统计信息。由于 DB2 优化器是基于代价计算来决定是否选择 MQT，准确地统计信息很重要。

（5）默认情况下，DB2 只考虑用系统维护 MQT 来优化查询。如果创建的 MQT 是用户维

护的，那么需要将注册变量 CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION 设置为 USER，如下所示：

```
db2 SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION USER
```

5.7 强大的数据压缩

在 I/O bound 的数据库系统中，压缩使得读取表、索引、临时表等对象只需更少的 I/O 次数，而且缓冲池中也能容纳更多的数据，从而提高缓冲池命中率，这对性能提高是非常有帮助的。除了压缩数据外，也可以压缩日志，从而减少归档时间；还可以压缩备份介质，从而缩短备份时间。

DB2 一直在压缩技术上不断改进。在 V9.1 中引入了行压缩技术，随后在 V9.5 中加入了 ADC（Automatic Dictionary Creation）特性，在 V9.7 中提供了对 XML 数据的压缩支持，在 V10.1 中更进一步提出了自适应压缩技术，可以根据不同的压缩样本灵活选择行压缩或者页压缩技术。

注意：CPU bound 的系统不应启用压缩

由于压缩和解压缩会带来一定的 CPU 开销，所以在 CPU bound 的系统中不应启用压缩。

5.7.1 行压缩深度剖析

行压缩是针对 V9.1 以前已有的值压缩技术提出的。值压缩是将表不同行、同一列的重复数据只保存一份，其他地方仅是引用而已。行压缩则是以行为数据压缩的基本单位，在压缩过程中，行不会被拆为列来处理。通过行压缩技术，用户能够为数据表建立一个表级别的数据字典，并使用该字典来压缩整个表中的全部数据。

压缩字典作为隐藏对象存储在数据库中，并且一般将它缓存在内存中以便快速访问。字典不会占用很多空间，即使对于非常大的表，压缩字典通常也只占用大约 100 KB 空间。从图 5-16 可以看到，字典包含数据行中重复的串，行压缩基于字典对每一行数据进行压缩。行是压缩的基本单位，这包含两点：

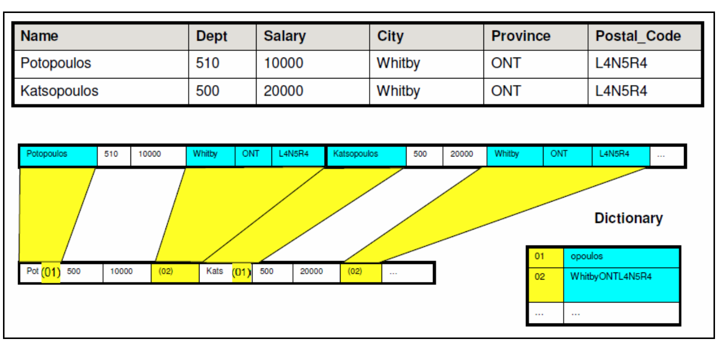


图 5-16　行压缩的技术原理

- 行不会被拆为列来处理；
- 不做跨行处理，一行数据压缩后还是完整的一行。

注意：分区表的独立压缩字典

表分区或 DPF 的每个分区都有自己独立的压缩字典。

1. 表压缩

那么如何对一个表启用压缩呢？很简单，在创建表的 CREATE TABLE 语句中使用 COMPRESS YES 选项；也可以在 ALTER TABLE 语句中使用 COMPRESS YES 随时对现有表启用压缩。例如，对 STORE_SALES 表启用压缩。

```
ALTER TABLE TPCD.STORE_SALES COMPRESS YES
```

但是启用压缩后还需要有创建字典的操作。在 DB2 V9.1 中，表重组是创建字典的唯一方式，但是对大表进行重组的开销较大，使用较为不便。变通的办法是，将一小部分数据先导入表中，在此之上进行表重组建立字典，然后后续进入表中的数据就会被自动压缩了。

但是，如果建立字典之后表中数据发生剧烈变化，原先的字典就有可能不能很好地压缩变化之后的数据，这时可以使用带有重置字典选项（RESETDICTIONARY）的表重组命令来重新创建压缩字典并用新字典压缩已有数据。例如：

```
REORG TABLE TPCD.DAILY_SALES RESETDICTIONARY;
```

如果启用表压缩后，表中已有压缩字典，重组操作默认情况下会保持字典不变，对数据进行重组，相当于使用 KEEPDICTIONARY 选项。

2. 索引压缩

当在建表时启用 COMPRESS YES 选项，默认情况下，该表上的索引也自动启用了压缩。当然，也可以在创建索引或者修改索引时显式指定是否启用压缩功能，命令如下：

```
CREATE INDEX … COMPRESS YES;
ALTER INDEX … COMPRESS YES;
```

注意：ALTER INDEX 语句

ALTER INDEX 语句不支持这样的更改：修改 CLUSTER 属性、索引升降序、修改组合索引的列等。如果需要对索引做这种修改，只能先删掉索引，然后重建。

3. 自动字典创建（ADC）

在 V9.1 中手工创建字典对 DBA 来说不是特别方便，所以在 V9.5 中增加了 ADC 特性。在启用行压缩后，压缩字典会在表中的数据增加到一定数量之后（默认是 2M）自动创建，随后表中新增的数据就会基于字典压缩。自动字典的建立简化了行压缩的使用，使得用户不再需要手工运行表重组操作了。

注意：ADC 与 REORG

有些读者可能对 ADC 的使用存在误区，认为有了 ADC 特性后就不需要手工 REORG 来维护压缩了。其实不然！建表时启用了压缩特性，当数据增加到一定量后，ADC 能自动地创建压缩字典，并压缩随后增加的数据。如果数据发生变化，压缩率下降时，则需要用 REORG 来重组表，建立新的压缩字典。

图 5-17 显示了自动创建压缩字典的过程：

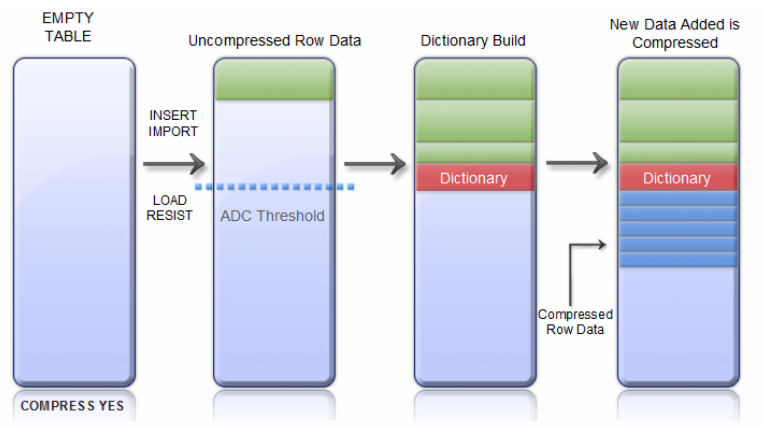


图 5-17　自动创建字典的技术原理

（1）创建表时指定了 COMPRESS YES 选项，由于是空表，没有创建压缩字典。

（2）刚开始时，插入（INSERT，IMPORT 等）到表中的数据并未压缩。

（3）随着更多数据插入到表中，当达到阈值（默认为 2M），就会自动触发字典的创建过程。DB2 基于已有的数据创建了压缩字典，并将压缩字典直接追加存储在表的数据后面，但是原有的数据并没有被压缩。

（4）此后，新增的数据基于这个字典而压缩。

DB2 只会对在创建字典之后才插入表中的行进行压缩。不会对创建字典之前就已存在的那些行进行压缩，除非更改那些行，或者使用 REORG 重建了字典。

由于 ADC 创建的字典仅基于很少的一部分数据，随着数据的变化，表的压缩率可能降低。这时，可以使用带有重置字典的选项对表进行重组。

5.7.2　V10 新特性：自适应压缩

行压缩的字典是静态的，除非使用带有 RESETDICTIONARY 选项进行重组操作，否则字典在创建之后不会随着数据的变化而发生变化。但是，随着表中数据的变化，使用静态的压缩字典可能使得表的压缩率降低；另外行压缩的字典是全局的，并不能反映数据的局部冗余特征，例如，一张表的某几处相邻的若干条数据相似度非常高，全局的压缩字典难以对这些局部的冗余数据达到最佳的压缩效果。

针对这两个问题，DB2 V10 提出了自适应压缩技术，它在原有行压缩的基础上，再叠加了页压缩。页压缩是动态的，它使用的字典是局部的，只针对其所在页的数据。当页中的数据达到一定程度时，页字典会自动被建立，并且已有数据会被自动压缩，当页中的数据发生变化后，

压缩率衰减到一定程度时，页字典会自动被重建。

如何使用自适应性压缩呢？在 DB2 V10 中引入了一个新关键字——ADAPTIVE，请看下列语法：

```
     .-COMPRESS NO---------------.
>---+---------------------------+--
     |                .-ADAPTIVE-. |
     '-COMPRESS YES-+----------+-'
                    '-STATIC---'
```

STATIC 关键字代表原有的行压缩方案，ADAPTIVE 代表对表启用新的自适应压缩，如果没有指定 STATIC 或者 ADAPTIVE 关键字时，默认启用自适应压缩。

例如，在建表时启用自适应压缩的语句如下：

```
create table t1(id int, name char(10), addr varchar(200))
  compress yes adaptive
```

在向一个启用了自适应压缩的表中插入数据时，如果表中存在行压缩字典，那么首先使用行压缩字典压缩这些数据，然后被放入数据页中；在一个数据页被填满之后，页压缩字典会自动被创建，页中的数据会被压缩。读取数据的解压也分为两步：先进行基于页压缩字典对数据页解压，然后再进行行压缩字典解压。当一个数据页上的数据发生剧烈变化导致页压缩的压缩率明显下降时，该页压缩字典会自动被重建，页中的数据会被重新压缩。

5.7.3　数据压缩实践与探索

由于用户数据的多样性，并不是所有数据都适合使用压缩处理。因此，在使用压缩以前，在不改变数据的情况下对压缩的效果进行评估是非常必要的，启用压缩后，也需要跟踪压缩率。也就是说，使用压缩技术包括下面四个步骤：

（1）启用压缩前，使用工具评估压缩率；

（2）对具有高压缩率的表启用压缩，随后进行重组；

（3）检查压缩后的效果，看实际节省了多少存储空间；

（4）随着系统的运行，监控压缩率，必要时重置字典重组表。

在 DB2 中，可以使用管理视图来估计启用压缩到底可以节省多少存储空间。先来了解一下管理视图的历史变迁，如表 5-3 所示。

表 5-3　压缩率有关的管理视图

估计压缩率的管理视图	
DB2 V9.1	没有管理视图，使用命令 INSPECT ROWCOMPESTIMATE
DB2 V9.5	ADMIN_GET_TAB_COMPRESS_INFO（第三个参数为'ESTIMATE')
DB2 V9.7	ADMIN_GET_TAB_COMPRESS_INFO_V97 的 ESTIMATE 模式
DB2 V10.1	SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO
报告压缩率	
DB2 V9.1	从 SYSCAT.TABLES 计算

（续表）

报告压缩率	
DB2 V9.5	ADMIN_GET_TAB_COMPRESS_INFO 的 REPORT 模式(第三个参数)
DB2 V9.7	ADMIN_GET_TAB_COMPRESS_INFO_V97 的 REPORT 模式
DB2 V10.1	SYSPROC.ADMIN_GET_TAB_DICTIONARY_INFO

注意：执行 RUNSTATS 命令

在执行这些压缩率有关的管理视图命令前，需要运行 RUNSTATS 命令获取最新的统计信息。

下面以 DB2 V9.7 为例，探讨一下如何结合管理视图来对表 TPCDS.CUSTOMER 以及其上的索引进行压缩。

1. 压缩表 TPCDS.CUSTOMER

第一步，使用 ADMIN_GET_TAB_COMPRESS_INFO_V97 的 ESTIMATE 模式（第 3 个参数）来估计 CUSTOMER 这个表的压缩率，如下所示，得到的压缩率为 60%。从输出中也可以看到，该管理视图会估计 XML 数据的压缩率，但是 CUSTOMER 表没有 XML 列，所以输出为 0。

```
    SELECT substr(TABSCHEMA,1,10) TABSCHEMA, substr(TABNAME, 1, 10) TABNAME,
COMPRESS_ATTR,COMPRESS_DICT_SIZE,ROWS_SAMPLED,PAGES_SAVED_PERCENT,AVG_COMPRESS_REC_LE
NGTH,OBJECT_TYPE FROM TABLE (SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO_V97('TPCDS', 'CUSTOMER',
'ESTIMATE'))"

    TABSCHEMA  TABNAME    COMPRESS_ATTR COMPRESS_DICT_SIZE   ROWS_SAMPLED
PAGES_SAVED_PERCENT AVG_COMPRESS_REC_LENGTH OBJECT_TYPE
    ---------- ---------- ------------- -------------------- ------------
------------------- ----------------------- -----------
    TPCDS      CUSTOMER   N                          37632       500000                  60        89 DATA
    TPCDS      CUSTOMER   N                              0            0                   0         0 XML

      2 record(s) selected.
```

第二步，使用 ALTER TABLE…COMPRESS YES 命令启用压缩，此时，表中的数据并不会压缩，接着使用 REORG 命令对表进行重组，并运行 RUNSTATS 命令更新统计信息，如下所示。

```
    alter table tpcds.customer compress yes;
    DB20000I  The SQL command completed successfully.
    reorg table tpcds.customer;
    DB20000I  The SQL command completed successfully.
    runstats on table tpcds.customer and indexes all;
    DB20000I  The SQL command completed successfully.
```

第三步，使用 ADMIN_GET_TAB_COMPRESS_INFO_V97 的 REPORT 模式来查看实际的压缩效果，得到的压缩率为 60%，与前面估计的一样。

```
    SELECT substr(TABSCHEMA,1,10) TABSCHEMA, substr(TABNAME, 1, 10) TABNAME,
COMPRESS_ATTR,COMPRESS_DICT_SIZE,ROWS_SAMPLED,PAGES_SAVED_PERCENT,AVG_COMPRESS_REC_LE
```

```
NGTH,OBJECT_TYPE FROM TABLE (SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO_V97('TPCDS','CUSTOMER',
'REPORT'))

    TABSCHEMA  TABNAME    COMPRESS_ATTR COMPRESS_DICT_SIZE   ROWS_SAMPLED
PAGES_SAVED_PERCENT AVG_COMPRESS_REC_LENGTH OBJECT_TYPE
    ---------- ---------- ------------- -------------------- ------------
------------------- ----------------------- -----------
    TPCDS      CUSTOMER   Y                37632       500000           60          89 DATA
    TPCDS      CUSTOMER   Y                    0            0            0           0 XML

      2 record(s) selected.
```

2. 压缩表 TPCDS.CUSTOMER 上的索引

第一步，与表压缩类似，通过管理视图 admin_get_index_compress_info 分析索引的压缩属性和压缩率，如下所示，表 TPCDS. CUSTOMER 上有 7 个索引，每个索引的压缩率并不相同，其中，CC_CURHDEMO、CFIRSTSHIPTODATE 和 C_SALESDATE 具有较好的压缩率，分别为 47%、52%和 52%。

```
    SELECT substr(indname,1,18) as indname, compress_attr, index_compressed,
pct_pages_saved, num_leaf_pages_saved FROM
TABLE(sysproc.admin_get_index_compress_info('', 'TPCDS', 'CUSTOMER', NULL, NULL)) AS t

    INDNAME           COMPRESS_ATTR INDEX_COMPRESSED PCT_PAGES_SAVED
NUM_LEAF_PAGES_SAVED
    ------------------ ------------- ---------------- ---------------
--------------------
    SQL120728090612660   N             N                      0                0
    CC_CURCDEMO          N             N                      0                0
    CC_CURHDEMO          N             N                     47              119
    C_CI                 N             N                     35              295
    C_CURADDR            N             N                      1                6
    C_FIRSTSHIPTODATE    N             N                     52              129
    C_SALESDATE          N             N                     52              129

      7 record(s) selected.
```

比较：COMPRESS_ATTR 和 INDEX_COMPRESSED 含义

COMPRESS_ATTR 表示索引的压缩属性是否启用，INDEX_COMPRESSED 表示索引是否被压缩。ALTER INDEX 启用压缩能改变前一个属性，而 REORG 后才表示索引真正被压缩了。

第二步，对这三个索引启用压缩，然后执行重组，并收集索引的统计信息，如下所示，不能重组单个索引，只能重组表上的所有索引。

```
    alter index tpcds.CC_CURHDEMO compress yes;
    DB20000I  The SQL command completed successfully.

    alter index tpcds.C_FIRSTSHIPTODATE compress yes;
```

```
DB20000I  The SQL command completed successfully.

alter index tpcds.C_SALESDATE compress yes;
DB20000I  The SQL command completed successfully.

reorg index tpcds.CC_CURHDEMO;
SQL0270N  Function not supported (Reason code = "89").  SQLSTATE=42997

reorg indexes all for table tpcds.customer;
DB20000I  The REORG command completed successfully.

runstats on table tpcds.customer for indexes all;
DB20000I  The RUNSTATS command completed successfully.
```

第三步，使用 admin_get_index_compress_info 查看索引的压缩属性和压缩率，如下所示，得到的实际压缩率与估计的差不多，CC_CURHDEMO、CFIRSTSHIPTODATE 和 C_SALESDATE 分别为 49%、52%和 52%。

```
    SELECT substr(indname,1,18) as indname, compress_attr, index_compressed,
pct_pages_saved, num_leaf_pages_saved FROM
TABLE(sysproc.admin_get_index_compress_info('', 'TPCDS', 'CUSTOMER', NULL, NULL)) AS t

    INDNAME            COMPRESS_ATTR INDEX_COMPRESSED PCT_PAGES_SAVED
NUM_LEAF_PAGES_SAVED
    ------------------ ------------- ---------------- ---------------
--------------------
    SQL120728090612660 N             N                               0                  0
    CC_CURCDEMO        N             N                               0                  0
    CC_CURHDEMO        Y             Y                              49                109
    C_CI               N             N                              35                295
    C_CURADDR          N             N                               1                  6
    C_FIRSTSHIPTODATE  Y             Y                              52                112
    C_SALESDATE        Y             Y                              52                112
      7 record(s) selected.
```

5.8 小结

数据库设计时需要特别考虑性能，一旦系统上线，后期的调整往往代价非常大，因此在一开始就需要做好设计规划。为了处理 TB 级的数据，本章首先从方法学谈起，介绍了高级设计技术包括数据库分区、表分区、MDC、MQT 和压缩技术的基本概念。随后，深入地剖析了一些高级技术的内幕以及如何在实践中灵活运用它们。相信读者真正掌握后，一定可以设计出一个高性能的数据库系统。

第6章 系统监控

数据库监控并不能带来直接的业务效益，在系统选型的时候很容易被忽略，但是对于 DBA 来说，缺少监控功能，就像缺少探照灯的行人在漆黑的夜里穿行在复杂的迷宫中，是不可能找到出路的。

类似的，当数据库系统的性能下降时，借助监控信息，可以根据数据库使用模式预计硬件要求，也可以分析各个应用程序或 SQL 语句的性能，跟踪索引和表的使用情况等来查找原因。本章首先介绍了系统监控和 DB2 数据库监控的工具，随后介绍了关键性能指标，最后读者可以在自己的运维工作中灵活运用这些知识监控和诊断常见的性能瓶颈。

6.1 由一则新闻想到的

最近看到一则新闻，2012 年 3 月 12 日 16 时许，长沙朝晖路附近，十几个青年正手持凶器追砍着几名男子。不料神兵天降，民警突然赶来并对犯罪嫌疑人进行了围捕，当场抓获涉案人员 19 名，随后还抓获网上逃犯 2 名。

你一定会感到奇怪，人民警察什么时候变得这么神奇了，接着往下看，你就不会感觉奇怪了，原来长沙市通过天网工程在全市安装了大量的监控摄像头，所以可以及时了解全市所发生的突发事件，从而及时出警，将恶性事件消灭在初始阶段。你可能又会问，这么多监控会不会有侵犯隐私的问题，这个就需要严格的法律法规来避免……

打住，这个和 DB2 的性能优化有什么关系？ DB2 的运行也需要 DBA 通过监控，尽早地发现问题，像长沙市民警一样，将问题消灭在初始阶段，这样才能保证系统良好的运行。那是不是有了监控手段，就能达到 DB2 良好运行的目的呢？正像没有法律法规保证的摄像头可能会侵犯隐私一样，不合理运用监控来解决问题，很有可能劳而无功，消耗了大量的系统资源，但是没有监控到解决问题所需要的信息。

那么如何才能合理运用监控呢？以招商引资为例，一个地方所能吸引投资的最大能力是由当地的资源所限制的，对于 DB2 来说，其最大的处理能力是由其硬件资源决定的，无论如何优化，也不可能超出硬件资源的处理能力，所以必须首先保证硬件资源能够满足业务对处理能力的需要。

有了硬件资源的保证，还需要优化招商引资的软环境，才能吸引更多的资金，对于 DB2 来说，操作系统的优化就像招商引资的外部环境，只有首先优化好操作系统，DB2 才能提供更强的处理能力。

有了以上条件，还必须对企业的工人进行技能的培训，才能最终生产出合格的产品，从而产生实际的效益，而对于 DB2 而言，也就是必须优化应用，调优访问数据库的 SQL 语句。

从以上可以总结出数据库性能调优的一般原则是硬件资源、操作系统、数据库系统以及访问数据库的应用。所以对于监控来说，在硬件资源足够的基础上，通过监控，发现或排除操作系统的问题，然后分析数据库系统的问题，最后解决数据库应用的问题，即调优访问数据库的 SQL 语句。

硬件资源的优化，在本书第 3 章的物理设计部分已有详细的介绍，接下来本章重点讲述系统监控的方法和 DB2 数据库监控的方法。

6.2 操作系统监控

本节讨论综合运用 DB2 和操作系统的监控工具进行 CPU、I/O 和内存监控的方法。当判断系统是否出现 CPU、I/O 或内存瓶颈时，需要利用如表 6-1 列出的常用工具得到系统状态数据

并进行分析。

表 6-1　操作系统监控的常用命令

	命令	操作系统	描述
CPU 监控	top； vmstat；sar –u；sar –q	LINUX/UNIX	检查 CPU 状态
	tprof	AIX	
I/O 监控	iostat；sar –d； sar –u；sar –b	LINUX/UNIX	检查操作系统的 I/O 状态
内存监控	vmstat； free； sar –b；top；	LINUX/UNIX	检查操作系统的内存利用情况
	lsps；svmon	AIX	
网络监控	netstat； ping	LINUX/UNIX WINDOWS	检查操作系统的网络状态
进程命令	ps	LINUX/UNIX	查看操作系统的进程相关信息
	taskmgr.exe	WINDOWS	
其他命令	lsof；ipcs；df； dd；nmon	LINUX/UNIX	查看进程打开文件信息、操作系统信号、存储等
	topas	AIX	

下面就重要的常用工具进行分别阐述。

6.2.1　CPU 监控

要重点关注 CPU 利用率、系统 CPU 时间（CPU 运行在内核态的时间）和用户 CPU 时间（CPU 运行在用户态的时间）。在 Unix 系统中，利用 sar 命令，可以看到每个 CPU 的利用率。下面是 sar 命令的一个典型输出：

```
   AIX nutmeg 3 5 00CD241F4C00    06/14/10

   System configuration: lcpu=4

11:51:33 cpu    %usr    %sys    %wio   %idle   physc
11:51:34  0       0       0       0     100     0.30
          1       1       1       1      98     0.69
          2       2       1       0      96     0.69
          3       0       0       0     100     0.31
          -       1       1       0      98     1.99
11:51:35  0       0       0       0     100     0.31
          1       0       0       0     100     0.69
          2       0       0       0     100     0.73
          3       0       0       0     100     0.31
          -       0       0       0     100     2.04
11:51:36  0       0       0       0     100     0.31
          1       0       0       0     100     0.69
          2       0       0       0     100     0.70
          3       0       0       0     100     0.31
          -       0       0       0     100     2.01
11:51:37  0       0       0       0     100     0.31
          1       0       0       0     100     0.69
          2       0       0       0     100     0.69
```

```
            3        0        0        0      100    0.31
            -        0        0        0      100    2.00

    Average  0        0        0        0      100    0.31
            1        0        0        0       99    0.69
            2        1        0        0       99    0.70
            3        0        0        0      100    0.31
            -        0        0        0       99    2.01
```

从这个输出可以看到每个 CPU 的利用率（%usr 指用户 CPU，%sys 指系统 CPU）情况和总的平均利用率（最后的 Average 栏）。

通过 vmstat，可以看到每个线程使用的用户 CPU（us 列）、系统 CPU（sy 列）、CPU 空闲（id 列）和 CPU 等待（wa 列）时间的比例：

```
vmstat 1 6
System configuration: lcpu=4 mem=15808MB
kthr   memory     page     faults      cpu
-----  -------    ------   --------    -----------
r  b  avm    fre    re  pi  po  fr  sr  cy  in   sy   cs  us  sy  id  wa
1  1  110996 763741 0   0   0   0   0   0 231    96   91  0   0   99  0
0  0  111002 763734 0   0   0   0   0   0 332  2365  179  0   1   99  0
0  0  111002 763734 0   0   0   0   0   0 330  2283  139  0   5   93  1
0  0  111002 763734 0   0   0   0   0   0 310  2212  153  0   0   99  0
1  0  111002 763734 0   0   0   0   0   0 314  2259  173  0   0   99  0
0  0  111002 763734 0   0   0   0   0   0 321  2261  177  0   1   99  0
```

当 CPU 的利用率（用户 CPU 加系统 CPU）长时间大于 95%，就可以认为系统出现了 CPU 瓶颈，需要根据具体情况继续往下分析。

在 AIX 系统上，输入 topas 命令，可以看到总的 CPU 利用率情况。进入 topas 界面后，输入两个“c”可以看到每个 CPU 状态，如下所示：

```
Topas Monitor for host:    maradona           EVENTS/QUEUES    FILE/TTY
Sat Dec 17 20:25:49 2011    Interval:  2

CPU   User%  Kern%  Wait%  Idle%               Reads        0  Rawin        0
2       0.3    0.5    0.0   99.1              Writes       2  Ttyout      927
0       0.3    0.5    0.0   99.2              Forks        0  Igets        0
4       0.1    0.1    0.0   99.7              Execs        0  Namei        1
6       0.0    0.1    0.0   99.9              Runqueue    1.5  Dirblk       0
5       0.0    0.0    0.0   99.9              Waitqueue   0.0
1       0.0    0.0    0.0  100.0                              MEMORY
3       0.0    0.1    0.0   99.9              PAGING           Real,MB   31872
                                               Faults       0  % Comp     37

Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out  Steals       0  % Noncomp  29
Total     1.6      8.0     2.5     0.5     1.1    PgspIn       0  % Client   29
                                                  PgspOut      0

Disk    Busy%     KBPS     TPS KB-Read KB-Writ  PageIn       0  PAGING SPACE
Total    0.0       0.0      0.0     0.0     0.0  PageOut      0  Size,MB   20480
```

```
FileSystem       KBPS     TPS KB-Read KB-Writ                % Free     99
Total            0.0       0.0    0.0    0.0

Name            PID   CPU%  PgSp  Owner          CliV2        0  WPAR Total    0
topas      10616944   0.1   2.5 iifvt14         SerV3        0  Press: "h"-help
dtgreet     3342804   0.0   1.3 root           CliV3        0          "q"-quit
```

从上面的 User%和 Kern%列可以分别看到用户和系统 CPU 利用率。

一般情况下，数据库系统是受 I/O 而不是 CPU 处理能力限制的（即一般为 I/O Bound，而不是 CPU Bound），所以 CPU 利用率通常不会很高。当 CPU 利用率长时间大于 95%，就可以认为系统出现了 CPU 瓶颈。此外，如果发现 CPU 利用率比正常情况下相同时段要高很多（比如平时 30%，现在 50%以上），用户也要通过进一步分析找出变化的原因。

6.2.2　I/O 监控

对于 DB2 来说，最慢、最花费时间的操作就是磁盘 I/O，其在整个系统中扮演着重要的角色。当整个系统的磁盘带宽不够，或者 I/O 请求集中在单一磁盘上，就容易产生磁盘 I/O 瓶颈。

利用下面的 sar 命令可以得到 I/O 状态报告（每 1 秒采样 1 次，连续采样 2 次）。

```
    # sar -d 1 2

AIX brave 1 6 0002568FD600    12/31/11

System configuration: lcpu=4 drives=3  mode=Capped

18:33:49   device     %busy    avque    r+w/s    Kbs/s   avwait   avserv

18:33:50    hdisk0      77      0.0      775    11040      0.2      2.1
           hdisk1      100     10.8      247    10928     33.5     12.2
           hdisk2      100      2.2      203     8256      8.8     14.3


18:33:51   hdisk0      78      0.0      793    11200      0.1      2.2
           hdisk1      74      0.1      864    12336      0.3      2.0
           hdisk2      95      2.0      305     8648      4.9      8.8



Average    hdisk0      77      0.0      784    11120      0.1      2.2
           hdisk1      87      5.5      555    11632     16.9      7.1
           hdisk2      97      2.1      254     8452      6.8     11.6
```

从上面示例输出中的%busy 列可以得到每个磁盘使用率，从 avwait 列可以得到平均磁盘等待次数。如果磁盘的使用率超过 80%（如上面监视结果中的磁盘 hdisk1 和 hdisk2），或者出现大量的磁盘等待，就说明存在磁盘瓶颈。这需要通过重新分布表，改变访问计划或者增加磁盘来消除瓶颈。

利用 iostat 也可以看到磁盘的忙闲状态，其中%tm_act 指示物理磁盘活动所占总时间的百分比（磁盘的带宽利用率），当利用率超过 70%时，进程将等待的时间会比完成 I/O 所必需的时间更长，这时就可能存在磁盘瓶颈，下面的监视结果显示磁盘的利用率较低，所以当前系统

不存在磁盘瓶颈(其后的参数 1 表示每秒种显示一次结果)：

```
# iostat 1

System configuration: lcpu=4 disk=4

tty:     tin        tout  avg-cpu:  % user    % sys     % idle    % iowait
          0.0       392.0              5.2      5.5       88.3      1.1

Disks:        % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk1           0.5      19.5       1.4   53437739  21482563
hdisk0           0.7      29.7       3.0   93086751  21482563
hdisk4           1.7     278.2       6.2   238584732  832883320
hdisk3           2.1     294.3       8.0   300653060  832883320
```

使用-d 选项，可以查看某个具体磁盘的状况：

```
# iostat -d hdisk1 1

System configuration: lcpu=4 disk=5

Disks:        % tm_act     Kbps      tps    Kb_read   Kb_wrtn
hdisk1           0.5      19.4       1.4   53437743  21490480
hdisk1           5.0      78.0       23.6        3633      3564
hdisk1           0.0       0.0       0.0         0         0
hdisk1           0.0       0.0       0.0         0         0
hdisk1           0.0       0.0       0.0         0         0
hdisk1           0.0       0.0       0.0         0         0
```

6.2.3 内存监控

内存作为 CPU 和磁盘之间的桥梁，对于 DB2 的性能起着至关重要的影响，合理地配置内存参数，能够有效地降低磁盘 I/O，减少锁升级等，从而极大提高系统性能，但是如果内存参数设置不合理，导致内存交换发生，反而会降低系统的整体性能。

在 Windows 系统中使用任务管理器，可以显示每个进程的内存使用情况。而在 Unix 或 Linux 系统中可以使用下面的 ps 命令得到每个进程的内存使用情况：

```
# ps gv | head -n 1; ps gv | egrep -v "RSS" | sort +6b -7 -n -r
  PID    TTY STAT  TIME PGIN  SIZE   RSS   LIM  TSIZ   TRS %CPU %MEM COMMAND
 15256      - A    64:15  755  2572  2888    xx  2356   316  0.9  0.0 /usr/lpp/
 22752      - A     0:08  261  1960  1980 32768   465    20  0.0  0.0 dtwm
 14654      - A     0:00  324  1932  1932    xx   198     0  0.0  0.0 /usr/sbin
 20700      - A     0:07  271  1868  1896 32768    95    28  0.0  0.0 /usr/dt/b
 20444      - A     0:03  203  1736  1824 32768   551    88  0.0  0.0 dtfile
 17602      - A     0:00  274   948  1644 32768   817   696  0.0  0.0 sendmail:
 13218      - A     0:00   74  1620  1620    xx   116     0  0.0  0.0 /usr/sbin
```

在 AIX 系统中，可以使用 svmon –G 命令，从输出结果中的 size 列和 virtual 列分别查看系统物理内存和在进程虚拟空间中分配的内存，从而看出是否存在物理内存不足的情况。

```
#svmon -G
             size       inuse       free         pin     virtual
```

```
memory       8159232     5706251     2522613     1756809  5584319
pg space     5242880     1689535

              work        pers        clnt       other
pin          1449315         0           0      307494
in use       4388042         0     1318209

PageSize   PoolSize       inuse        pgsp         pin      virtual
s    4 KB         -     2205803      430415      400313      1206031
m   64 KB         -      218778       78695       84781       273643
```

用下面的 vmstat 命令，可以查看是否出现了内存瓶颈：

```
# vmstat 1 4

System Configuration: lcpu=4 mem=4096MB
kthr    memory              page              faults       cpu
----- ----------- ------------------------ ------------ -----------
 r  b   avm   fre  re   pi   po   fr   sr   cy  in   sy   cs   us sy id wa
 1  2 136583  127    0   4   57   44   92    0  345 2223 605  30 40 29 1
 2  7 136587  118    0   2  230  0    245   0  329 3451 526  20 37 10 33
 1  6 136587  157    0   3   67  0    678   0  334 3304 536  25 32 20 23
 3  8 136587  111    0   5   61  0    693   0  329 3341 511  19 26 35 20
```

fre 列表示空闲的物理内存页面数，而 avm 列表示系统工作所需的页面数。如果系统工作所需要的超出物理内存就会发生换页。从上面的输出可以看出，系统存在很多的页换出（page out，在 po 列），I/O 等待时间（wa）也很长，空闲的内存页（fre 列）却很少。这都说明系统存在物理内存不足的情况。

利用下面的 sar 命令，可以从 fault 列看出是否出现了大量的页面错误（page fault，指进程访问的页面不在内存中，需要从磁盘读入）。如果 page fault 过多，可能需要增加用于文件缓存的内存或增加系统的物理内存。

```
# sar -r 1 5
System Configuration: lcpu=4 mem=4096MB

06:18:05   slots  cycle/s  fault/s  odio/s
06:18:06 1048052    0.00    387.25   0.00
06:18:07 1048052    0.00    112.97   0.00
06:18:08 1048052    0.00    45.00   79.21
06:18:09 1048052    0.00    216.00    0.00
06:18:10 1048052    0.00    8.00      0.00

Average  1048052        0      79      16
```

db2mtrk 工具也可以被用来查看内存使用状态。使用下面的命令可以查看数据库中各种堆包括缓冲池的使用状态：

```
[db2inst1@db2 ~]$db2mtrk -d -v
Tracking Memory on: 2011/01/02 at 16:46:58

Memory for database: FLIGHTDB
```

```
Backup/Restore/Util Heap is of size 196608 bytes
Package Cache is of size 7929856 bytes
Other Memory is of size 196608 bytes
Catalog Cache Heap is of size 917504 bytes
Buffer Pool Heap (1) is of size 330498048 bytes
Buffer Pool Heap (System 32k buffer pool) is of size 655360 bytes
Buffer Pool Heap (System 16k buffer pool) is of size 393216 bytes
Buffer Pool Heap (System 8k buffer pool) is of size 262144 bytes
Buffer Pool Heap (System 4k buffer pool) is of size 196608 bytes
Shared Sort Heap is of size 262144 bytes
Lock Manager Heap is of size 28442624 bytes
Database Heap is of size 39845888 bytes
Application Heap (60) is of size 65536 bytes
Application Heap (59) is of size 65536 bytes
Application Heap (58) is of size 65536 bytes
Application Heap (57) is of size 65536 bytes
Application Heap (56) is of size 65536 bytes
Application Heap (55) is of size 196608 bytes
Application Heap (54) is of size 65536 bytes
Application Heap (53) is of size 65536 bytes
Application Heap (52) is of size 65536 bytes
Application Heap (51) is of size 131072 bytes
Applications Shared Heap is of size 1966080 bytes
Total: 412614656 bytes
```

6.2.4 网络监控

网络问题的调整非常困难，因为有太多的变化因素相互影响，往往只有调整了系统中其他的方面之后发现仍然存在性能问题，才会注意到是网络的性能造成的瓶颈。可以通过 netstat –s 命令来帮助诊断问题：

```
# netstat -s
.........
Tcp:
    23057 active connections openings
    743 passive connection openings
    1119 failed connection attempts
    914 connection resets received
    32 connections established
    1153134 segments received
    928463 segments send out
    21456 segments retransmited
    0 bad segments received.
    2115 resets sent
.........
TcpExt:
    53 invalid SYN cookies received
    7 resets received for embryonic SYN_RECV sockets
    6579 TCP sockets finished time wait in fast timer
    29 packets rejects in established connections because of timestamp
    15053 delayed acks sent
    5 delayed acks further delayed because of locked socket
    Quick ack mode was activated 7395 times
```

```
    16189 packets directly queued to recvmsg prequeue.
    430150 bytes directly in process context from backlog
    9384212 bytes directly received in process context from prequeue
    799853 packet headers predicted
    6902 packets header predicted and directly queued to user
    50683 acknowledgments not containing data payload received
    24955 predicted acknowledgments
    12 times recovered from packet loss due to fast retransmit
    204 times recovered from packet loss by selective acknowledgements
    11 congestion windows recovered without slow start by DSACK
    2569 congestion windows recovered without slow start after partial ack
    TCPLostRetransmit: 4
    20 timeouts after reno fast retransmit
    30 timeouts after SACK recovery
    218 timeouts in loss state
    248 fast retransmits
    45 forward retransmits
    844 retransmits in slow start
    8542 other TCP timeouts
    8 classic Reno fast retransmits failed
    135 SACK retransmits failed
    7231 DSACKs sent for old packets
    496 DSACKs sent for out of order packets
    1816 DSACKs received
    526 connections reset due to unexpected data
    207 connections reset due to early user close
    1680 connections aborted due to timeout
    TCPSACKDiscard: 1
    TCPDSACKIgnoredNoUndo: 280
    TCPSackShifted: 9
    TCPSackMerged: 31
    TCPSackShiftFallback: 1689
    TCPRcvCoalesce: 160837
........
```

如果（segments retransmitted/segments send out）的结果大于 10%，则极有可能遇到了严重的吞吐问题，几乎可以肯定要增加更多的带宽、或者减少进出网络的流量大小。

6.3 数据库监控

DB2 提供了多种多样的监控手段，主要包括快照监视器（Snapshot Monitor）、事件监视器（Event Monitor）、监视器表函数（Monitoring Table Functions）、管理视图（Administrative Views）、db2pd 以及 db2top 等。工欲善其事必先利其器，DBA 需要根据具体的需求选用合适的工具，做到物尽其用。

6.3.1 快照监视器

DB2 中最常用也最直接的监控工具就是快照监视器。快照监视器可以让你获得当前时间点的数据库状态信息。一般来说，快照监视器的开销不大（大约占用 3%~6%的系统开销）。如果

每隔一定时间间隔获取快照，那么快照在观察和预测潜在问题方面也很有用。

要想获取快照信息，必须先打开快照开关。快照信息包括 BUFFERPOOL（缓冲池）、LOCK（锁）、SORT（排序）、STATEMENT（SQL 语句）、TABLE（表）、TIMESTAMP（时间戳）和 UOW（事务）等，它们需要用各自的开关打开。表 6-2 列出了快照开关名称、其对应的数据库管理器（DBM）参数和对应的快照信息。

表 6-2　快照开关及其对应的 DBM 参数和快照信息

快照开关	DBM 参数	对应的快照信息
BUFFERPOOL	DFT_MON_BUFPOOL	收集缓冲池读写次数和读写所花费的总时间等信息
TIMESTAMP	DFT_MON_TIMESTAMP	收集各种时间相关信息
UOW	DFT_MON_UOW	收集事务的开始/结束时间，完成状态等信息
SORT	DFT_MON_SORT	收集排序所使用的堆大小等信息
STATEMENT	DFT_MON_STMT	收集 SQL 语句开始/结束时间，SQL 语句文本等信息
TABLE	DFT_MON_TABLE	收集从表中的读写行数等信息
LOCK	DFT_MON_LOCK	收集发生的锁等待次数、死锁次数等信息

如果想在每次重新连接数据库时就自动开始收集某项快照信息，那么需要将对应的 DBM 参数设为 ON。例如下面的命令可以将缓冲池快照信息对应的 DBM 参数 DFT_MON_BUFPOOL 设为 ON。这样以后每次重新连接时，缓冲池快照信息开关都是打开的。

```
db2 connect to FLIGHTDB
db2 update dbm cfg using DFT_MON_BUFPOOL ON
db2 terminate
```

也可以在每次连接数据库时，使用 update monitor switches 显式打开或关闭快照开关。命令的语法如下：

```
db2 update monitor switches using [bufferpool | lock | sort | statement | uow | table
| timestamp] [on | off]
```

例如，下面的命令将锁快照信息开关打开。此后锁快照信息将被持续收集，直到数据库连接结束并执行 db2 terminate，或者使用 update monitor switches 命令关闭快照开关。

```
db2 update monitor switches using lock on
```

你可以使用下面的命令查询各个快照开关是否被打开了：

```
=> db2 get monitor switches

          Monitor Recording Switches

Switch list for db partition number 0
Buffer Pool Activity Information  (BUFFERPOOL) = OFF
Lock Information                        (LOCK) = OFF
Sorting Information                     (SORT) = OFF
SQL Statement Information          (STATEMENT) = OFF
Table Activity Information             (TABLE) = ON  2012-11-07 15:33:10.888675
Take Timestamp Information         (TIMESTAMP) = ON  2012-11-07 15:20:45.884026
Unit of Work Information                 (UOW) = OFF
```

监视器开关打开后，可以使用 get snapshot 命令得到快照信息，基本语法为（<dbname>为数据库名）：

```
db2 get snapshot for [bufferpools | locks | tables | dynamic sql | applications tablespaces
| database | all] on db <dbname>
```

获取的快照信息按照元素类型分为以下种类，如表 6-3 所示。

表 6-3　快照信息的元素类型

计数器	计算活动的发生次数。在监视期间，计数器会自动增长，通常计数器元素可以复位
标尺	指示某个项的当前值。根据数据库活动的不同，标尺值会增加或减小（例如，挂起的锁定数）。标尺元素不能复位
水位标记	指示从开始监视以来元素所达到的极限值（最大或最小值）。水位标记元素不能复位
信息	提供参考类型的监视活动详细信息。这可以包括诸如分区名称、别名和路径详细信息这类的项。信息元素不能复位
时间戳记	通过提供自 1970 年 1 月 1 日后经历的秒数和微秒数，以指示发生的日期和时间。时间戳记元素不能复位
时间	返回执行活动所耗用的秒数和微秒数，某些时间元素可以复位

例如，可以使用下面的命令得到数据库 FLIGHTDB 上 SQL 语句的快照信息，从中可以看出每条 SQL 语句的执行次数、总的执行时间等信息：

```
=> db2 get snapshot for dynamic sql on FLIGHTDB

     Dynamic SQL Snapshot Result

 Database name                = FLIGHTDB

 Database path                = F:\DB2\NODE0000\SQL00001\ ← [信息]

 Number of executions              = 79551  ←执行次数[计数器]
 Number of compilations            = 1
 Worst preparation time (ms)        = 29 ←[水位标记]
 Best preparation time (ms)         = 29
 Internal rows deleted             = 0  ←[标尺]
 Internal rows inserted            = 0
 Rows read                       = 10000
 Internal rows updated             = 0
 Rows written                     = 0
 Statement sorts                   = 0
 Statement sort overflows           = 0
 Total sort time                   = 0
 Buffer pool data logical reads      = 51
 Buffer pool data physical reads     = 0
 Buffer pool temporary data logical reads   = 0
 Buffer pool temporary data physical reads  = 0
 Buffer pool index logical reads     = 79551
 Buffer pool index physical reads     = 0
 Buffer pool temporary index logical reads  = 0
 Buffer pool temporary index physical reads = 0
 Buffer pool xda logical reads      = 0
```

```
 Buffer pool xda physical reads    = 0
 Buffer pool temporary xda logical reads   = 0
 Buffer pool temporary xda physical reads  = 0
 Total execution time (sec.microsec)= 5.694087 ←总的执行时间
 Total user cpu time (sec.microsec) = 3.328125
 Total system cpu time (sec.microsec)= 0.234375
 Statement text = SELECT COUNT(*) FROM HOLDINGQUEUE WHERE QUEUE = ? FOR READ ONLY
......
```

使用下面的命令，可以得到数据库 FLIGHTDB 的数据库快照信息，从中可以看出数据库的基本状态，包括锁、缓冲池和排序等：

```
=> db2 get snapshot for database on FLIGHTDB

            Database Snapshot

Database name                          = FLIGHTDB
………
Locks held currently                    = 3990
Lock waits                             = 28573          ┐
Time database waited on locks (ms)         = 22579766    │
Lock list memory in use (Bytes)           = 491264      │
Deadlocks detected                     = 1311          │
Lock escalations                       = 0            ├ 锁相关信息
Exclusive lock escalations               = 0          │
Agents currently waiting on locks          = 0        │
Lock Timeouts                          = 0            ┘
Number of indoubt transactions            = 0

Total Private Sort heap allocated         = 0           ┐
Total Shared Sort heap allocated          = 36          │
Shared Sort heap high water mark          = 1137064     │
Post threshold sorts (shared memory)       = 0          ├ 排序相关信息
Total sorts                            = 10567804       │
Total sort time (ms)                     = 2825803      │
Sort overflows                          = 221840        │
Active sorts                           = 2              ┘

Buffer pool data logical reads           = 2789551727    ┐
Buffer pool data physical reads          = 75176050      │
Buffer pool temporary data logical reads   = 10683339    │
Buffer pool temporary data physical reads  = 0           ├ 缓冲池相关信息
Asynchronous pool data page reads         = 664237       │
Buffer pool data writes                 = 1190909        │
Asynchronous pool data page writes        = 1188306      │
Buffer pool index logical reads          = 1771670606    ┘
......
```

使用下面的命令，可以得到数据库 FLIGHTDB 汇总的快照信息：

```
db2 get snapsot for all on FLIGHTDB
```

所有的快照信息都可以通过执行 RESET MONITOR 命令归零。这个命令的基本语法是（其

中<dbname>为数据库名称）：

```
db2 reset monitor [for [[database | db] <dbname>] | all]
```

6.3.2　事件监视器

事件监视器收集特定事件发生时的监控数据，例如事务提交、SQL 语句执行完成或者死锁等等。事件监视器必须先由用户创建，可以手动打开和关闭。一旦打开后，事件监视器就会不断捕获相关事件的信息。你可以创建多个事件监视器，然后根据需要打开和关闭它们，灵活确定监控的时间和范围。表 6-4 列出了事件监视器所监控的事件以及收集数据的内容和时间点。

表 6-4　监控事件以及收集数据的内容和时间点

事件监视器类型	所收集的信息	收集时间点
DATABASE	数据库信息	所有数据库连接断开
BUFFERPOOLS	缓冲池信息、预取程序（Prefetcher）信息等	所有数据库连接断开
TABLESPACES	表空间 I/O 信息、与表空间相关的缓冲池信息等	所有数据库连接断开
TABLES	对每个表读取或写入的行数	所有数据库连接断开
CONNECTIONS	所有应用程序级别的计数器	数据库连接结束
STATEMENTS	SQL 语句执行结束	SQL 语句开始和停止的时间、 CPU 时间、SQL 文本、SQL 语句的返回码等
DEADLOCKS	SQL 语句以及各语句具有的锁列表	发生死锁时
ACTIVITIES	SQL 语句和编译环境等信息	服务类、工作负载或工作类中所执行活动完成时
STATISTICS	根据每个服务类、工作负载或工作类内的活动数据计算而来的统计信息	每隔预定时间
THRESHOLD VIOLATIONS	阈值违例信息	检测到阈值违例时
UNIT OF WORK	事务相关的信息	在事务完成时

事件监控器用于捕获快照监控器捕获不到的信息，例如死锁循环的相关信息。实际上，DB2 中有一种称为死锁检测器的后台代理程序，会按预定义的时间间隔被唤醒，在锁定系统中扫描死锁循环。如果检测到死锁循环，死锁探测器会随机选择、回滚并终止此循环中的一个事务，以释放此事务获得的锁，使其余事务能够继续执行。这些事件无法被快照监控器捕获，因为当你获取快照时，死锁循环可能已经被检测到并解除了。而监控锁事件的事件监控器却可以在死锁循环被检测到时立即激活，并记录相关的重要信息。除了处理死锁等特殊事件，事件监视器还能记录查询语句的开始和结束时间，CPU 使用时间等一系列信息。

下面来看一下如何创建事件监视器。事件监视器需要通过执行 CREATE EVENT MONITOR

语句创建。此语句的基本语法是：

```
>>-CREATE EVENT MONITOR--event-monitor-name--FOR---------------->

   .-,-----------------------------------------------------------.
   V                                                             |
>----+-+-DATABASE------------------------------------------------+-+-+-->
     | +-TABLES--------------------------------------------------+ |
     | +-LOCKING--+-------------------------------------------+-+ |
     | +-TABLESPACES---------------------------------------------+ |
     | '-BUFFERPOOLS---------------------------------------------' |
     '-+-CONNECTIONS--+--+--------------------------+---------'
       +-STATEMENTS---+  '-WHERE--| event-condition |-'

>--WRITE TO--+-TABLE--| evm-group-info |---------+--●----------->
             +-PIPE--pipe-name------------------+
             '-FILE--path-name--| file-options |-'

   .-MANUALSTART-.
>--+-------------+--●------------------------------------------->
   '-AUTOSTART---'

>--+----------------------------------------+--●---------------->
   '-ON DBPARTITIONNUM--db-partition-number-'

   .-LOCAL--.
>--+--------+--●-----------------------------------------------><
   '-GLOBAL-'
```

例如，可以使用下面的语句创建 SQL 语句事件监视器 MYEM，它将记录应用程序 APP1 以用户 MYUSER 访问数据库时执行的语句信息。这里指定了“write to table” 选项，这使得 DB2 在创建 MYEM 的同时，会在当前用户模式下面创建一些表，以便将事件发生时收集的信息写入这些表中。

```
db2 "create event monitor MYEM for statements where appl_name= 'APP1' and auth_id=
'MYUSER' write to table"
```

通过下面的 list tables 命令，可以看到创建的用于存放事件信息的三张表：CONNHEADER_MYEM（事件监视相关的连接信息）、CONTROL_MYEM（事件监视相关的控制信息）和 STMT_MYEM（事件监视器所收集的语句信息）。注意，这些表的表名都以事件监视器名称即 MYEM 结尾。

```
    => db2 list tables |grep MYEM
CONNHEADER_MYEM               MYUSER        T     2012-11-07-5.20.45.214101
CONTROL_MYEM                  MYUSER        T     2012-11-07-5.20.45.335244
STMT_MYEM                     MYUSER        T     2012-11-07-5.20.45.738973
```

使用 set event monitor 语句打开 MYEM：

```
db2 set event monitor MYEM state 1
```

这样，当监控的应用程序连接数据库、执行 SQL 语句或者最后断开数据库时，事件监视器都会进行监控并将收集到的 SQL 语句信息存入 STMT_MYEM 等三张表中。可以使用下面的语句关闭 MYEM：

```
db2 set event monitor MYEM state 0
```

然后，通过查询表 STMT_MYEM，可以得到每条 SQL 语句的各种信息，包括开始执行和结束时间、CPU 时间以及排序时间等。例如，利用下面的查询，可以得到其中耗费 CPU 时间最多的语句：

```
    =>db2 " select STMT_TEXT from STMT_MYEM order by SYSTEM_CPU_TIME + USER_CPU_TIME fetch
first row only"

    STMT_TEXT
    ---------------------------------------------------
    select t.tabname, i.indname from syscat.tables t, syscat.indexes i  where t.tabname
= i.tabname and coalesce(t.index_tbspace, t.tbspace)=3
```

需要注意的是，当数据库关闭后，事件监视器也会被关闭。如果在创建事件监视器时指定了 AUTOSTART 选项，数据库重新启动时，事件监视器将自动打开。

在上面创建事件监视器时指定将收集的信息写入表。其实也可以将信息写入指定的管道或者文件。如果指定写入文件，需要指定一个已存在的文件夹。信息收集完毕后，使用 db2evmon 格式化收集到的信息。具体方法如下。

首先创建一个名为 CONNEM 的事件监视器，用于监控数据库连接。这里指定了“write to file”选项，将收集的事件信息写入目录“/home/myuser/eventmonitor”下的文件。

```
create event monitor CONNEM for connections write to file
'/home/myuser/eventmonitor'
```

经过打开监视器、收集信息和最后关闭的过程后，可以在目录“/home/myuser/eventmonitor”下看到存放事件信息的文件。使用下面的 db2evmon 命令，就可以得到格式化的事件信息，从中可以看出每个连接的应用程序名、用户名和连接时间等信息。

```
#查看/home/myuser/eventmonitor 目录下的文件
=> ls -ltr /home/myuser/eventmonitor
total 24
-rw-r-----    1 myuser pdxdb2          31 Nov 08 01:03 db2event.ctl
-rw-r--r--    1 myuser pdxdb2        4543 Nov 08 01:03 00000000.evt

#使用 db2evmon 对事件信息进行格式化，-path 选项指定事件信息存放的目录
=>db2evmon -path '/home/myuser/eventmonitor'
Reading /home/myuser/eventmonitor/00000000.evt ...
--------------------------------------------------------------------------
                        EVENT LOG HEADER
  Event Monitor name: CONNEM
  Server Product ID: SQL09058
  Version of event monitor data: 9
  Byte order: BIG ENDIAN
  Number of nodes in db2 instance: 1
```

```
  Codepage of database: 819
  Territory code of database: 1
  Server instance name: myuser
--------------------------------------------------------------------------
  Database Name: FLIGHTDB
  Database Path: /home/myuser/myuser /NODE0000/SQL00001/
  First connection timestamp: 11/08/2012 23:49:50.572444
  Event Monitor Start time:   11/08/2012 01:03:21.246989
--------------------------------------------------------------------------

1) Connection Header Event ...
  Appl Handle: 57
  Appl Id: *LOCAL.myuser.121110806573 7
  Appl Seq number: 00003
  DRDA AS Correlation Token: *LOCAL.myuser.111221065737
  Program Name    : db2bp    ←连接对应的应用程序名
  Authorization Id: MYUSER  ←连接对应的用户名
  Execution Id    : myuser
  Codepage Id: 819
  Territory code: 1
  Client Process Id: 668596
  Client Database Alias: FLIGHTDB
  Client Product Id: SQL09058
  Client Platform: Unknown
  Client Communication Protocol: Local
  Client Network Name: matthaus
  Connect timestamp: 11/08/2012 00:57:37.921527 ←连接建立的时间
......
```

6.3.3 监视器表函数

为了让用户能以 SQL 语句的形式方便地获得监控信息，DB2 从 V8.1 版本开始，提供了监视器表函数（Monitoring Table Functions）。在 V8.1 中，DB2 提供了函数名以“SNAPSHOT_”开头的表函数；在 V9.1 中，又以一组“SNAP_GET_”开头的函数代替。这里重点介绍以“SNAP_GET_”开头的这组表函数。

下面的查询可以列出 DB2 支持的所有以“SNAP_GET_”开头的监视器表函数：

```
db2 "select substr(funcname,1,50) from syscat.functions
     where funcname like 'SNAP_GET_%'"
```

表 6-5 列出了常用的监控对象、对应的表函数名和返回的信息。

表 6-5 监视器表函数的监控对象、表函数名和返回的信息

监控对象	表函数名	返回的信息
应用程（Application）	SNAP_GET_APPL	应用程序快照信息（如读取行数、更新行数等）
	SNAP_GET_APPL_INFO	应用程序基本信息（如应用程序名称、状态等）
缓冲池（Bufferpool）	SNAP_GET_BP	缓冲池快照信息

（续表）

监控对象	表函数名	返回的信息
数据库（Database）	SNAP_GET_DB	数据库快照信息
	SNAP_GET_DBM	数据库管理器快照信息
	SNAP_GET_HADR	HADR 快照
	SNAP_GET_STORAGE_PATHS	数据库的存储路径及其所在文件系统的可用空间信息
	SNAP_GET_STO_PATHS	数据库的存储路径
日志（Log）	SNAP_GET_DETAILLOG	日志信息
SQL 语句(SQL Statement）	SNAP_GET_DYN_SQL	动态 SQL 语句快照信息
	SNAP_GET_STMT	所有 SQL 语句的快照信息（包括动态和静态 SQL）
锁（Lock）	SNAP_GET_LOCK	锁快照信息
	SNAP_GET_LOCKWAIT	锁等待快照信息
表（Table）	SNAP_GET_TAB	表的快照信息（如读取行数、写入行数等）
表空间（Tablespace）	SNAP_GET_CONTAINER	表空间容器信息
	SNAP_GET_TBSP	表空间快照信息
实用程序（Utility）	SNAP_GET_UTIL	实用程序信息（类型、优先级等）
	SNAP_GET_UTIL_PROGRESS	实用程序的状态信息
	SNAP_GET_TAB_REORG	表重组信息（表名、重组类型、状态等）

注意，要利用上述监视器表函数获取快照监控信息，你需要先打开相应的实例级快照监控开关（如 DFT_MON_BUFPOOL）。

使用监视器表函数可以方便获得各种快照信息。例如，针对下面的查询，可以利用表函数 SNAP_GET_TAB 获得数据库 FLIGHTDB 的表读取和写入（包括插入和更新）的行数。 参数“－1”表示对当前连接的数据库分区（如果为多分区数据库）获取此快照。从 ROWS_READ 和 ROWS_WRITTEN 列可以分别看到每个表读取和写入的行数。

```
=> db2 "select TABNAME, ROWS_READ, ROWS_WRITTEN from table(SNAP_GET_TAB(FLIGHTDB, -1))"

TABNAME                          ROWS_READ           ROWS_WRITTEN
------------------------- -------------------- --------------------
SYSTABLES                          56                   0
FLIGHTDB                              1377332              229389
......
```

从 DB2 V9.7 开始，可以通过监视表函数来代替传统系统监视器命令对 DB2 进行监控。这些表函数增强了收集和查看系统活动和数据对象数据的能力。而且与传统监视器相比，这种轻量级的表函数对系统的影响更小。DB2 的监控表函数分为系统监控表函数、活动监控表函数和对象监控表函数。

1. 系统监控表函数

下面是可以用来访问系统级别监视信息的表函数：

- MON_GET_SERVICE_SUBCLASS 和 MON_GET_SERVICE_SUBCLASS_DETAILS。
- MON_GET_WORKLOAD 和 MON_GET_WORKLOAD_DETAILS。
- MON_GET_CONNECTION 和 MON_GET_CONNECTION_DETAILS。
- MON_GET_UNIT_OF_WORK 和 MON_GET_UNIT_OF_WORK_DETAILS。

这组表函数使你能够得到特定详尽程度的监控信息。它们是成对存在的：一个表函数用于对常用数据进行访问，另一个表函数用于对全部可用的监控信息进行访问。

缺省情况下，这些表函数将自动收集系统监控信息。你可以使用下列方法更改缺省设置：

（1）利用数据库配置参数 mon_req_metrics 指定所有服务类的最低收集级别。此参数可以设为：NONE（不收集监控信息）和 BASE（收集所有监控信息）。

（2）利用 CREATE/ALTER SERVICE CLASS 语句的 COLLECT REQUEST METRICS 子句指定服务类的收集级别。使用此设置来将特定服务类的收集级别增大为高于 mon_req_metrics 的级别。这里也是可以将级别设为 NONE 和 BASE。

例如，如果只收集部分服务类的系统监视信息，请执行下列操作，将数据库配置参数 mon_req_metrics 设置为 NONE。

```
=> db2 update db cfg using MON_REQ_METRICS NONE
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

（3）对于每个希望搜集监控信息的服务类，将 CREATE/ALTER SERVICE CLASS 语句的 COLLECT REQUEST METRICS 子句设置为 BASE。

```
=> db2 alter service class s1 collect request metrics base
DB20000I  The SQL command completed successfully.
```

2. 活动监控表函数

活动是指 SQL 语句的执行行为。活动监视表函数用于得到 SQL 语句执行的各个方面的数据，以及 SQL 语句文本等其他信息。活动相关的信息将随着活动的进行在内存中累积。而与 SQL 语句活动直接相关的信息，还随着 SQL 语句的每次执行在高速缓冲中累积。

使用表函数 MON_GET_ACTIVITY_DETAILS 和 MON_GET_PKG_CACHE_STMT 可以访问活动的当前数据。MON_GET_ACTIVITY_DETAILS 以 XML 格式返回活动数据。MON_GET_PKG_CACHE_STMT 返回特定 SQL 语句所有执行的归总数据。

例如，使用下面的查询可以查看消耗 CPU 时间最多的前 10 个动态 SQL 语句。将参数“D”改为“S”，可以得到对应静态 SQL 语句的结果，改为 NULL 则得到对应所有 SQL 语句的结果：

```
select STMT_TEXT
from table(mon_get_pkg_cache_stmt('D', NULL, NULL, -2)) as T
order by total_cpu_time
fetch first 10 rows only

STMT_TEXT
----------------------------------------------------------------------------
```

```
SELECT TABNAME FROM SYSCAT.TABLES  WHERE TABNAME='HMON_COLLECTION' ……
SELECT TABNAME FROM SYSCAT.TABLES  WHERE TABNAME='HMON_ATM_INFO' AND  ……
UPDATE SYSTOOLS.HMON_ATM_INFO AS ATM SET STATS_FLAG = 'N' WHERE (ATM.SCHEMA, …….
SELECT POLICY FROM SYSTOOLS.POLICY WHERE MED='DB2TableMaintenanceMED' ……
```

使用下面的查询可以查看引用热表<HotTableName>的动态 SQL，把 pool_data_p_reads、pool_data_writes 和 num_executions 列放入 order by 子句，分别可以得到读、写和执行次数最多的 10 个动态 SQL 语句。将参数“D”改为“S”，可以得到对应静态 SQL 语句的结果，改为 NULL 则得到对应所有 SQL 语句的结果：

```
select *
from table(mon_get_pkg_cache_stmt('D', NULL, NULL, -2)) as T
where stmt_text like '%<HotTableName>%'
order by …
fetch first 10 rows only
```

使用下面的查询可以查看访问索引页最多的 10 个动态 SQL 语句，<TableNameForHotIndex>为热索引对应的数据表名。将参数“D”改为“S”，可以得到对应静态 SQL 语句的结果，改为 NULL 则得到对应所有 SQL 语句的结果：

```
select *
from table(mon_get_pkg_cache_stmt('D', NULL, NULL, -2)) as T
where stmt_text like '%<TableNameForHotIndex>%'
order by pool_index_p_reads desc
fetch first 10 rows only
```

使用下面的查询可以查看被返回的数据行数与被访问数据行数比例最低的 10 个 SQL 语句，包括静态和动态 SQL，这也是执行成本较高的 SQL 语句：

```
select *
from table(mon_get_pkg_cache_stmt(NULL, NULL, NULL, -2)) as T
where rows_read <> 0
order by rows_returned/rows_read asc
fetch first 10 rows only
```

缺省情况下，DB2 会自动收集活动监视信息。你可以使用下列设置中的一个或全部来更改缺省设置：

（1）利用 mon_act_metrics 数据库配置参数指定所有工作负载的最低收集级别，默认值为 BASE，即自动收集。也可以将 mon_act_metrics 设置为 NONE，从而不收集活动监视信息。

```
=> db2 update db cfg using mon_act_metrics none
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

（2）利用 CREATE/ALTER WORKLOAD 语句的 COLLECT ACTIVITY METRICS 子句指定给定工作负载的收集级别（高于对所有工作负载设置的最低收集级别）。

（3）将 CREATE/ALTER WORKLOAD 语句的 COLLECT ACTIVITY METRICS 子句设置为 BASE。缺省情况下，其他工作负载的值为 NONE。

```
=> db2 create workload w1 system_user('DB2INST1') collect activity metrics base
DB20000I  The SQL command completed successfully.
```

3. 数据对象监视表函数

提供操作数据对象（即表、索引、缓冲池、表空间和容器）的相关信息。对于每种对象类型，都有一组不同的监控信息。每当数据对象被处理时，该对象的监控信息都将递增。例如，在从特定表中读取行时，该表的“读取行数”值将递增。

可以使用下列表函数访问数据对象的当前信息，这些表函数返回当前数据，但你无法访问数据对象的历史数据。

- MON_GET_BUFFERPOOL；
- MON_GET_TABLESPAC；
- MON_GET_CONTAINER；
- MON_GET_TABLE；
- MON_GET_INDEX。

例如，可以使用下面查询查看缓冲池命中率：

```
select * from table(mon_get_bufferpool('',-2)) as metrics
```

下面的查询可以查看被访问数据行数最多的表：

```
select tabschema, tabname, sum(rows_read) as total_rows_read
from table(mon_get_table('','',-2)) as T
group by tabschema, tabname
order by total_rows_read
fetch first row only"
```

数据对象监控信息是否收集，由数据库配置参数 mon_obj_metrics 来定义。如果它的值为 BASE，则收集数据对象监控信息；如果值为 NONE 则不收集，例如下面的语句设定其值为 NONE。

```
=> db2 update db cfg using mon_obj_metrics none
DB20000I  The UPDATE DATABASE CONFIGURATION command completed successfully.
```

注意：有两个表函数不受 mon_obj_metrics 的影响

无论 mon_obj_metrics 参数怎么设置，DB2 都将搜集表函数 MON_GET_TABLE 和 MON_GET_INDEX 相关的监控数据。

6.3.4 管理视图

从 V9.1 开始，DB2 提供了一套管理视图（Administrative Views），用于方便获取数据库监控信息。利用这些管理视图，一方面可以获得原来需要用快照命令或监视器表函数才能获得的信息（例如，利用视图 SNAP_BP 可以获得缓冲池的快照信息）；另一方面，也可以得到在这些信息之上计算出来的结果（例如，利用视图 BP_HITRATIO 可以直接获得各个缓冲池的命中率），这些结果是由视图通过快照信息计算而来的。

这些管理视图的模式都为 SYSIBMADM，通过 db2 list tables for schema SYSIBMADM 可以获取所有管理视图的列表：

```
=> db2 list tables for schema SYSIBMADM

Table/View                    Schema          Type  Creation time
------------------------- ------------------- --------------------------
ADMINTABINFO                  SYSIBMADM       V     2012-11-03-13.55.52.309004
APPLICATIONS                  SYSIBMADM       V     2012-11-03-13.55.52.309032
APPL_PERFORMANCE              SYSIBMADM       V     2012-11-03-13.55.52.309036
AUTHORIZATIONIDS              SYSIBMADM       V     2012-11-03-13.55.52.293011
BP_HITRATIO                   SYSIBMADM       V     2012-11-03-13.55.52.324003
BP_READ_IO                    SYSIBMADM       V     2012-11-03-13.55.52.324007
BP_WRITE_IO                   SYSIBMADM       V     2012-11-03-13.55.52.324011
CONTACTGROUPS                 SYSIBMADM       V     2012-11-03-13.55.52.309012
CONTACTS                      SYSIBMADM       V     2012-11-03-13.55.52.309016
CONTAINER_UTILIZATION         SYSIBMADM       V     2012-11-03-13.55.52.324015
DBCFG                         SYSIBMADM       V     2012-11-03-13.55.52.309020
DBMCFG                        SYSIBMADM       V     2012-11-03-13.55.52.309024
DBPATHS                       SYSIBMADM       V     2012-11-03-13.55.52.309028
DB_HISTORY                    SYSIBMADM       V     2012-11-03-13.55.52.293003
......
```

注意，和监视器表函数类似，要利用管理视图获取快照监控信息，需要先打开相应的实例级快照监控开关。

如果系统的用户 CPU 时间消耗过多时，可以从管理视图 sysibmadm.snapdyn_sql 中按耗费用户 CPU 时间排列所有的动态 SQL 语句，通常只需要关注返回结果的前几行动态 SQL 语句：

```
select STMT_TEXT
from sysibmadm.snapdyn_sql
order by total_usr_cpu_time desc

STMT_TEXT
----------------------------------------------------------------------------
select * from DB2INST1.LOCATION q1 where  ( q1."LOCATIONCODE" = ?)  and
( q1."LOCATIONTYPE" = 'AD')  and  ( q1."STARTDATE" <= ?)  and  ( q1."ENDDATE" >= ?)  and
( q1."ACTIVERECORD" = ?)
select * from DB2INST1.ORGANISATION q1 where  ( q1."ORGANISATIONCODE" = ?)  and
( q1."ORGANISATIONTYPE" = 'OPR')  and  ( q1."STARTDATE" <= ?)  and  ( q1."ENDDATE" >= ?)
and  ( q1."ACTIVERECORD" = 'Y')
select * from DB2INST1.ROUTE q1 where  ( q1."ROUTECODE" = ?)  and
( q1."ROUTEDESCRIPTOR" = ?)  and  ( q1."STARTDATE" <= ?)  and  ( q1."ENDDATE" >= ?)
Select * from DB2INST1.ROUTE q1 where  ( q1."ROUTECODE" = ?)  and
( q1."ROUTEDESCRIPTOR" = ?)  and  (? >=  q1."STARTDATE")  and  (? <=  q1."ENDDATE")
......
```

6.3.5 db2pd

db2pd 工具是独立运行于 DB2 引擎之外的诊断工具。它可以查看数据库的各种状态信息，主要用于对特定问题的监控和故障诊断。这里重点介绍如何使用 db2pd 进行一般性的监控。

与前面几节提到的快照、事件监视器等工具相比，使用 db2pd 进行监控有以下优点：

- db2pd 直接从 DB2 的内存中读出信息，不需要连接数据库，所以不会像快照或事件监视器那样占用数据库引擎资源。

- db2pd 收集信息时不会请求任何锁，所以它获取信息的速度很快。
- db2pd 提供的信息比其他的监控方式更底层也更丰富。比如数据库挂起时，DB2 已经无法接受外部连接，这时只能通过 db2pd 获取诊断信息。另外，对于新的监控信息，DB2 一般先通过 db2pd 提供，随后再由其他途径提供。例如，有些索引、表和 SQL 信息，先是通过 db2pd 的某些选项提供，随后从 DB2 V9.7 开始相应的监视器表函数也提供类似的查看功能，如表 6-6 所示：

表 6-6　db2pd 选项和表函数的对应

db2pd 选项	对应的新监视器表函数
db2pd –tcbstats index	mon_get_index
db2pd –tcbstats	mon_get_table
db2pd –static	mon_get_pkg_cache_stmt

注意：db2pd 收集的信息有时不准确

在 db2pd 收集信息时，有可能会检索到正在被更改的数据，这时返回的数据可能不精确。另外，如果遇到正在更改的内存，会导致 db2pd 的输出信息中包含这样的消息："Changing data structure forced command termination"，这时只需重新执行 db2pd 命令即可。

db2pd 有很多选项，提供关于事务、表空间、表统计信息、动态 SQL 以及数据库配置等信息。用户只要输入 db2pd -everything，就会列出能提供的所有信息。如果想限定输出的范围，可以使用表 6-7 中的选项：

表 6-7　db2pd 用于限制输出范围的选项

选项	信息
Ins	输出实例相关信息
db <dbname>	输出数据库<dbname>的相关信息
Alldb	输出所有数据库信息
dbp <partitionnum>	输出数据库分区<partitionnum>的信息
Alldbp	输出所有数据库分区的信息
Osinfo	输出操作系统相关信息

db2pd 用于获得数据库对象信息的选项可以分为实例和数据库两个层次。实例层次的选项有 agents、fcm 等，如表 6-8 所示：

表 6-8　db2pd 实例层次的选项

选项	信息
Agents	输出 DB2 代理（Agent）的相关信息
Fcm	输出多分区数据库中的快速通信管理器（FCM）信息
Mempools	输出内存池相关信息

（续表）

选项	信息
Dbmcfg	输出 DBM 配置信息，等同于 get dbm cfg 命令，但输出格式稍有不同
Utilities	输出正在执行的实用程序的相关信息

数据库层次的选项有 applications、transactions 等，如表 6-9 所示：

表 6-9　db2pd 数据库层次的选项

选项	信息
Applications	输出应用程序信息，同时还有很多相关子选项用于更精确的输出某个应用程序的相关信息
Transactions	输出数据库的事务信息
Bufferpools	输出数据库的缓冲池信息
Logs	输出数据库的日志信息
Locks	输出数据库的锁信息
Reorgs	输出数据库的表重组信息

下面举一些使用 db2pd 进行监控的实例。

使用 db2pd –app，可以看到当前正在运行的应用程序的状态。例如，从下面的输出可以看到，句柄为 57 的应用程序处于 UOW-Executing 状态，即正在执行事务。

```
=> db2pd -app -db FLIGHTDB

Database Partition 0 -- Database FLIGHTDB -- Active -- Up 2 days 23:04:11

Applications:
Address            AppHandl [nod-index] NumAgents  CoorTid    Status        L-AnchID
0x000000069C6692A0 57       [000-00701] 1          3808       UOW-Waiting   236
0x000000069C664940 735      [000-00735] 1          4876       UOW-Waiting   159
0x000000069F1D3DC0 1376     [000-01376] 1          4296       UOW-Waiting   33
0x000000069F1D3CC0 1372     [000-01372] 1          2972       UOW-Waiting   137
......
```

接下来，利用 db2pd –activestatements 可以得到正在执行的语句的 AnchorID 为 236，这个 ID 可以作为 SQL 语句的标识符在包缓冲中找到对应语句。

```
=>  db2pd -activestatements -db FLIGHTDB

Database Partition 0 -- Database FLIGHTDB -- Active -- Up 2 days 23:04:26

Active Statement List:
Address            AppHandl [nod-index] UOW-ID     StmtID     AnchID StmtUID    EffISO

0x0700000053742420 57       [000-00057] 9          1          236    1          0
```

使用 db2pd –dyn 输出包缓冲中的所有动态 SQL，根据 AnchorID 可以进一步查到 SQL 语句

的文本，这就是句柄为 57 的应用程序正在执行的语句。

```
    =>db2pd -dyn -db FLIGHTDB
    Database Partition 0 -- Database FLIGHTDB -- Active -- Up 2 days 23:04:40

    ......
    Dynamic SQL Statements:
    Address            AnchID StmtUID    NumEnv     NumVar     NumRef     NumExe     Text
    0x0700000053687AA0 175    1          1          1          3          3
    select distinct MESSAGESOURCE from   DB2INST1.DATAINTEGRITYMESSAGE as basetable
where lower(basetable.PROCESSNAME) LIKE '%%' FOR READ ONLY

    0x070000005373FDA0 236    1          1          1          2          2
    select * from sysibmadm.snapdyn_sql order by total_usr_cpu_time desc
```

db2pd 是一个非常强大的监控工具，它提供了观察数据库内部状态的途径，运维人员应该熟练掌握这一利器。

6.3.6 db2top

db2top 是一个强大的、基于单一视图的监控工具。使用 db2top 监控数据库，类似于 Unix 操作系统中的 top 命令监控主机状态，它在运行过程中会不断刷新，从而提供实时的监控信息。

db2top 支持 AIX、Linux 和 Solaris 等三种平台，并且在 DB2 V8 FP17、DB2 V9.1 FP6 以及 DB2 V9.5 FP2 以上版本才可以使用。

图 6-1 是 db2top 的初始界面（输入命令“db2top –d <数据库名>”后可以看到）。其中列出了各种选项（如 d – Database，用于获得数据库整体状态），输入这些选项（例如输入“d”）就可以查看相应的具体监控信息了。

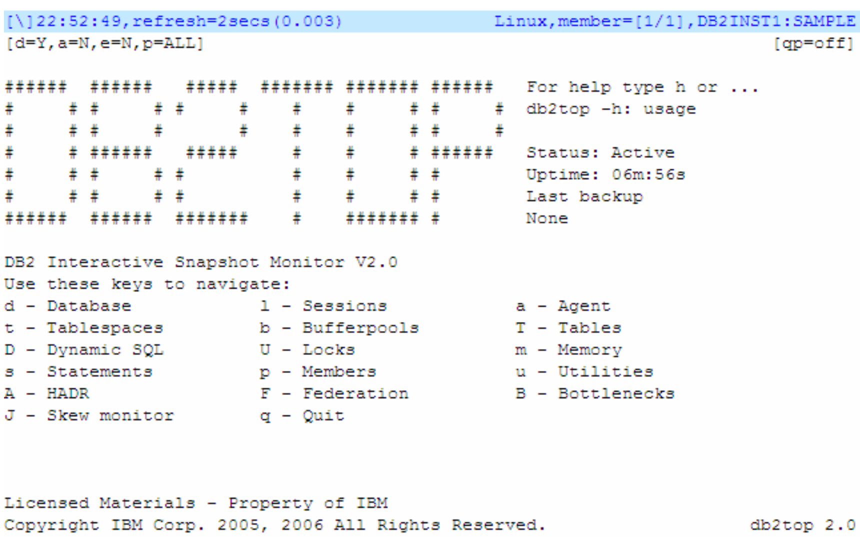


图 6-1 db2top 初始界面

通过 db2top，可以实时地查看 DB2 中的性能瓶颈。在初始界面中输入选项“B”，可以看到性能瓶颈相关的信息，如图 6-2 所示。其中 SessionCpu 行表明当前哪个应用程序占用了最多的 CPU 时间，而 Memory 行则显示了哪个应用程序占用了最大的内存。

```
[\]22:47:17,refresh=2secs(0.002)            Inactive,member=[1/1],DB2INST1:SAMPLE
[d=Y,a=N,e=N,p=ALL]                                                      [qp=off]

    |              |            25%|          50%|          75%|         100%|
    |wait lock ms  |                                                          |
    |sort ms       |                                                          |
    |bp r/w ms     |                                                          |
    |async r/w ms  |                                                          |
    |pref wait ms  |                                                          |
    |dir r/w ms    |----------------------------------------------------------|

     Server       Top      Resource Resource         Application
     Resource     Agent    Usage    Value            Name
  -- ------------ -------- -------- ---------------- --------------------
  => Cpu              N/A       0%                0 N/A
  => SessionCpu       557   60.18%         0.291921 db2bp
  => IO r/w           557  100.00%               29 db2bp
  => Memory           557   23.40%           704.0K db2bp
  => Locks            557  100.00%                5 db2bp
  => Sorts            557  100.00%               79 db2bp
  => Sort Time        N/A       0%                0 N/A
Quit: q, Help: h          db2v97.sample.gicp.com                    db2top 2.0
```

图 6-2　db2top 的系统瓶颈分析界面

由于 db2top 显示结果随着 DB2 实际运行的环境变化而更新,有时需要获得显示的内容很困难，这个时候，就可以以后台方式运行而将结果写入文件，下面的语句可以以后台的方式获取动态 SQL，并将结果写入文件 top.sql 中。其中-b 表示后台执行，而子命令 D 表示动态 SQL，-m 参数指定以该收集模式运行多少分钟。

```
=>$ db2top -d sample -b D -o top.sql -m 5
DB2 active: 07m:01s
Writing output to top.sql
```

6.4　关键 SQL 语句监控

通常一个应用中包含成千上万条 SQL，但是影响应用性能的往往是少数关键 SQL（Costly SQL），这些关键 SQL 也是潜在的不良 SQL，如果能够快速地识别它们，并利用第 9 章的 SQL 优化知识进行针对性的优化，往往能够起到事半功倍的效果。

6.4.1　监控最耗费用户 CPU 时间的动态 SQL

通过下面的查询可以从 snapdyn_sql 视图返回耗费用户 CPU 时间(total_usr_cpu_time)最多的 5 条动态 SQL 语句，对于 CPU 瓶颈的系统，可以有针对性地优化这些 SQL：

```
select STMT_TEXT
from sysibmadm.snapdyn_sql
order by total_usr_cpu_time desc
fetch first 5 rows only
```

```
STMT_TEXT
---------------------------------------------------------------------------
select t.tabname, i.indname from syscat.tables t, syscat.indexes i  where t.tabname
= i.tabname and coalesce(t.index_tbspace, t.tbspace)=3
......
```

6.4.2 监控访问热点对象执行次数最多的 SQL

针对某些 I/O 瓶颈的问题，需要找出访问这些热点对象的 SQL 语句进行调整，从而消除热点，而热点对象包含热表和热索引。

当确定某个表例如“FLIGHTHISTORY”为热表时，就可以使用下面的语句从 snapdyn_sql 视图中找出访问该表执行次数（num_executions）最多的 5 条语句及其执行次数。并从整体上进行优化，以消除访问的热点：

```
select stmt_text, num_executions
from sysibmadm.snapdyn_sql
where stmt_text like '%FLIGHTHISTORY%'
order by num_executions desc
fetch first 5 rows only

sql_text                                                          num_executions
---------------------------------------------------- …     -----------------
select  q1."ACTIVERECORD",  q1."AIRPORTCATEGORYCAF",      14644
......
```

使用下面的语句可以从视图 **snapdyn_sql** 查询访问索引页最多的动态 SQL 语句，其中 <TableNameForHotIndex>为热索引（被频繁访问的索引）对应的数据表名：

```
select stmt_text, num_executions
from sysibmadm.snapdyn_sql
where stmt_text like '%<TableNameForHotIndex>%'
order by pool_index_p_reads desc
fetch first row only
```

6.4.3 监控运行时间最长的 SQL

可以使用下面的查询从视图 long_running_sql 监控正在执行的运行时间(**elapsed_time_min**)最长 SQL 语句：

```
select *
from sysibmadm.long_running_sql
order by elapsed_time_min desc
fetch first row only
```

6.4.4 监控执行次数最多的 SQL

有时候，单个 SQL 语句的执行时间并不很长，但是由于其执行次数较多，也有可能会成为系统的瓶颈，可以从视图 top_dynamic_sql 中识别这样的 SQL，通过如下查询，可以获得执行次数最多的 5 条 SQL（stmt_ext）及其执行次数（num_executions）：

```
select stmt_text,num_executions
from sysibmadm.top_dynamic_sql
order by num_executions desc
fetch first 5 rows only
```

6.4.5 监控排序次数最多的 SQL

当由于过多的排序造成 CPU 使用过高时，可以通过如下查询，从视图 top_dynamic_sql 中获得排序次数（stmt_sorts）最多的 5 条 SQL 语句（stmt_text）进行调整：

```
select stmt_text,stmt_sorts
from sysibmadm.top_dynamic_sql
order by stmt_sorts desc
fetch first 5 rows only
```

6.5 关键性能指标（KPI）

现在感冒发烧很普遍，医生通常会要求验血，现在的血检报告，即使是非专业人士也能简单地阅读，因为各项指标的参考值都会附在检验结果后面，关键参数偏离参考值就表示这项结果可能会有问题，人体也就可能有这样或者那样的疾病。对于数据库来说，也存在这样的关键性能指标，我们称之为 KPI（Key Performance Indicator）。

6.5.1 缓冲区命中率

公式：（逻辑读－物理读）/ 逻辑读

指标：数据缓冲区命中率（Data Bufferpool Hit Ratio）：> 95%

索引缓冲区命中率（Index Bufferpool Hit Ratio）：> 98%

数据库中数据的访问都需要经过缓冲池：读的数据需要先读到缓冲池才能提交给应用，写的数据也要先写到缓冲池才能进行 I/O。I/O 操作作为数据库中最慢的环节，对数据库性能起着至关重要的影响，如果数据能够直接从缓冲区中获取，将大大减少 I/O 操作。缓冲池命中率可以从管理视图中方便地获取，从下面的结果来看，缓冲池 BPBCPI003 和 BPBCPI004 数据缓冲区命中率偏低，需要进一步分析，对于命中率为零的缓冲池通常是因为没有使用到，这个时候就需要进一步分析是否需保存该缓冲池。

```
    db2 "SELECT SUBSTR(BP_NAME,0,15) AS BP_NAME,TOTAL_HIT_RATIO_PERCENT || '%' AS
TOTAL_HIT_RATIO,DATA_HIT_RATIO_PERCENT || '%' AS DATA_HIT_RATIO, INDEX_HIT_RATIO_PERCENT
|| '%' AS INDEX_HIT_RATIO FROM SYSIBMADM.BP_HITRATIO WHERE BP_NAME NOT LIKE 'IBMSYS%'"

    BP_NAME          TOTAL_HIT_RATIO DATA_HIT_RATIO INDEX_HIT_RATIO
    ---------------- --------------- -------------- ---------------
    IBMDEFAULTBP               99.98          99.99           99.87
    BPBCPD001                   0.00           0.00               -
    BPBCPD002                  97.05          97.05               -
    BPBCPD003                  99.99          99.99               -
    BPBCPD004                  99.99          99.99           99.91
```

```
    BPBCPI001                  0.00          0.00              -
    BPBCPI002                 99.96         99.35          99.96
    BPBCPI003                 99.99         69.74          99.99
    BPBCPI004                 99.99         40.98          99.99
    BPBCPDRPT                  0.00          0.00              -
```

6.5.2 包缓冲区命中率

公式：1－包缓冲区插入次数 / 包缓冲区查询次数

指标：包缓冲区命中率（Package Cache Hit Ratio）：> 98%

当数据库引擎接收一条查询请求的时候，首先会从包缓冲中查找有没有已经被编译好的包可以直接运行，如果查询到已经存在，就可以直接运行而不用再次编译从而节省 SQL 语句编译的时间。通常在编写应用的时候应该使用绑定变量，这样能够提高包缓冲区的命中率。下面的运行结果显示包缓冲区命中率 96.88%低于 98%，这时就需要进一步分析原因，是由于缓冲池不足还是由于应用中过多的查询没有使用绑定变量查询造成的命中率偏低，如果是前者，可以参考第 7 章来调整数据库参数，如果是后者，可以参考第 9 章来调整 SQL 语句。

```
    db2 "SELECT CAST(((1-PKG_CACHE_INSERTS/DOUBLE(PKG_CACHE_LOOKUPS))*100) AS
NUMERIC(5,2) )|| '%' AS PACKAGE_HIT_RATIO FROM SYSIBMADM.SNAPDB"

    PACKAGE_HIT_RATIO
    ------------------------------------------
    96.88%
```

6.5.3 编目缓冲区命中率

公式： 1 －编目缓冲区插入次数/编目缓冲区查询次数

指标：编目缓冲区命中率（Catalog Cache Hit Ratio）：> 98%

编目信息也就是数据库本身的元信息，几乎每一个数据库操作都需要查询编目表，因此 DB2 单独开辟了一块内存空间，用于存储编目信息，如果编目缓冲过小，那么每次想要查找编目数据时，就要从编目表空间中查找，这样会导致系统性能下降。因些一般需要将 catalogcache_sz 设置到合适的值，使编目缓冲区命中率保持在 98%以上。通过执行下面的查询语句，就可以得到编目缓冲区的命中率，从当前的执行结果可以得出当前数据库的编目缓冲区命中率为 98.53%，是达到要求的。

```
    db2 "SELECT CAST(((1-CAT_CACHE_INSERTS/DOUBLE(CAT_CACHE_LOOKUPS))*100) AS
NUMERIC(5,2) )|| '%' AS CATALOG_HIT_RATIO FROM SYSIBMADM.SNAPDB"

    CATALOG_HIT_RATIO
    ------------------------------------------
    98.53%
```

6.5.4 缓冲区读写平均响应时间

公式：缓冲区读取或写入时间/物理读取或写入次数

指标：平均读写时间：< 5 ms

该指标反映 I/O 的性能，如果监控的结果值超出了指标，就需要结合操作系统 I/O 监控工

具来作进一步分析，通过执行下面的查询，可以得出当前系统的缓冲区读写的平均响应时间是满足性能要求的：

```
    db2 "SELECT R.BP_NAME,AVERAGE_READ_TIME_MS,AVERAGE_WRITE_TIME_MS FROM
SYSIBMADM.BP_READ_IO AS R,SYSIBMADM.BP_WRITE_IO AS W WHERE
R.SNAPSHOT_TIMESTAMP=W.SNAPSHOT_TIMESTAMP AND R.BP_NAME=W.BP_NAME AND R.BP_NAME NOT LIKE
'IBMSYS%'"

    BP_NAME        AVERAGE_READ_TIME_MS AVERAGE_WRITE_TIME_MS
    -------------- -------------------- ---------------------
    IBMDEFAULTBP                  1                  0
    BPBCPD001                    0                  -
    BPBCPD002                    0                  0
    BPBCPD003                    4                  0
    BPBCPD004                    1                  0
    BPBCPI001                    0                  -
```

6.5.5　日志写入速度

公式：日志写入时间/日志写次数

指标：< 3ms

对于 OLTP 系统来说，日志的写入速度对应用程序的性能有着决定性的影响。通过下面的查询，可以得到当前的日志写入速度为 0.1437ms，因此是达到要求的：

```
    db2 "SELECT (LOG_WRITE_TIME_NS/1000000)/NUM_LOG_WRITE_IO AS AVERAGE_LOG_WRITE_MS
FROM SYSIBMADM.SNAPDB"

    AVERAGE_LOG_WRITE_MS
    ---------------------
    0.1437
```

6.5.6　锁升级及死锁

指标：（Lock Escalations and Deadlocks）：仅有少量的锁升级或死锁

锁升级或者死锁会极大地影响系统的并发性能，因此应该将 LOCKLIST 和 MAXLOCK 参数调整到足够大，尽量避免锁等待、锁升级，并调整应用避免死锁的发生，从而使数据库的并发性达到最大。从下面的结果可以看出系统发生了 1311 次死锁，所以需要通过使用死锁监控器来进一步分析死锁的原因。

```
    db2 "SELECT LOCK_ESCALS,DEADLOCKS FROM SYSIBMADM.SNAPDB"

    LOCK_ESCALS          DEADLOCKS
    -------------------- --------------------
             0                   1311
```

6.5.7　排序溢出率

公式：排序溢出次数/总排序次数

指标：（Sort Overflow / Total Sort）：< 1%

排序溢出就是当排序内存不够时，数据需要使用临时表空间进行排序。因此排序溢出时会产生大量 I/O 操作，从而影响 SQL 语句的执行性能。通过下面的查询，可以看到当前系统的排序溢出率为 7.83%，超出指标很多，这个时候，就需要分析是由于排序堆内存不足还是由于 SQL 语句不佳造成的，进而分别采取措施：

```
    db2 "SELECT CONCAT(CHAR(CAST((SORT_OVERFLOWS/DOUBLE(TOTAL_SORT_TIME))*100 AS
NUMERIC(4,2))),'%') AS SORT_OVERFLOW_RATIO FROM SYSIBMADM.SNAPDB"

    SORT_OVERFLOW_RATIO
    -------------------
    07.83 %
```

6.5.8 数据、索引页异步清除比例

公式：异步写入/总写入

指标：（Async Writes/Total Writes）：> 95%

该指标代表着页面清除进程（线程）是否能够有效地将脏页在后台写入磁盘。当该清除进程一直处于空闲状态或者由于缓冲池大小不合适，造成 db2 无法找到一个干净的页面，就会发生同步写入，相对于异步写入，同步写入会对数据读取性能造成问题。通过以下语句查询可以看出当前系统的异步清除比例是否合格：

```
    db2 "SELECT SUBSTR(BP_NAME,0,15) AS BP_NAME, PERCENT_WRITES_ASYNC FROM
SYSIBMADM.BP_WRITE_IO WHERE BP_NAME NOT LIKE 'IBMSYS%'"

    BP_NAME          PERCENT_WRITES_ASYNC
    ---------------- --------------------
    IBMDEFAULTBP                       97
    BPBCPD001                          96
    BPBCPD002                          98
```

6.5.9 脏页偷取次数

公式：脏页偷取触发次数

指标：脏页偷取次数（Dirty page steal cleaner triggers）非常低

脏页偷取过多则意味着页清除器的工作能力不足，需要调整页清除器相关的设置，如果系统中页清除器一直很空闲，则可以通过调节 softmax 与 chngps_thres 来让其更频繁地工作，但是如果页清除器一直非常繁忙，则需要增加页清除器的数量，页清除器的初始数量通常与 CPU 的数量相同。使用如下查询，可以看出当前系统由于页清除器的工作能力不足，从而发生了 2216 次的脏面偷取，因些需要增加页清除器的数目：

```
    db2 "SELECT POOL_DRTY_PG_STEAL_CLNS FROM SYSIBMADM.SNAPDB WHERE
POOL_DRTY_PG_STEAL_CLNS>0"

    POOL_DRTY_PG_STEAL_CLNS
    -----------------------
                       2216
```

6.6　融会贯通——监控的最佳实践

通过上面章节的学习，相信您已经掌握了 DB2 监控工具的使用，也就是掌握战争中武器的使用方法，但是如果不学习战略战术，还是难以取得战争的胜利。对于监控来讲，分为常规监控和对于特定问题的监控，本章主要关注性能问题的监控。

性能问题往往分为两大类：影响整个系统的问题和只影响部分系统的问题。但是它们的界限通常不是那么明显，例如特定应用或不良的 SQL 语句通常只会影响部分系统，但是有时这些特定应用造成了系统的瓶颈，就会影响整个系统。

第 1 章提到了 5 种常见的瓶颈，其中网络瓶颈在 6.2.4 节已经详细讲述过了，下面重点讲述 CPU、磁盘、内存、懒惰系统 4 种瓶颈的监控和诊断方法。

6.6.1　CPU 瓶颈问题的监控与诊断

CPU 瓶颈表现在两个方面：

（1）所有 CPU 都饱和意味着这个系统上的所有处理器都繁忙。根据 vmstat 和 perfmon 的报告，一般的标准是计算用户 CPU 时间和系统 CPU 时间。CPU 利用率如果长时间都超过 95%就被认为是饱和的。

（2）单 CPU 饱和意味着在一个处理器上的负载达到了饱和，但是其他处理器却部分或完全空闲。虽然还有可用的 CPU 能力，系统却无法消费，因此，应用程序或语句的速度取决于这一个处理器内核的繁忙程度。

同样也应该考虑到用户模式和系统模式上不同的 CPU 消耗。一种是处理器正在运行操作系统内核以外的软件，比如 DB2 应用程序或者中间件，导致用户 CPU 时间增多。另一种是在运行操作系统内核的时候系统 CPU 增多。这两个在数量上是分开显示的，并且根据它们之间的分布情况，可以有助于找到导致 CPU 瓶颈的原因。用户和系统 CPU 时间比率在 3:1 到 4:1 之间是正常的。如果在有瓶颈的系统中，用户和系统时间比率高于这个区间，就需要调查用户 CPU 时间为什么会增加了。

使用 db2pd –edus 命令，从 USR 和 SYS 列可以看到每个 DB2 代理程序耗费的用户和系统 CPU 时间，单位为秒：

```
Database Member 0 -- Active -- Up 0 days 00:42:54 -- Date 2012-11-17-16.39.36.374898

List of all EDUs for database member 0

db2sysc PID: 42720
db2wdog PID: 42718
db2acd  PID: 42739

EDU ID    TID     Kernel TID     EDU Name                 USR (s)        SYS (s)
==================================================================================
305      1398540843843         db2agntdp (DB2V102 ) 0    0.830000     0.750000
304      1398571243842         db2agntdp (DB2V102 ) 0    1.770000     1.730000
```

```
302      1398532043840      db2agent (DB2V102) 0     3.650000     5.650000
303      1398501643841      db2agntdp (DB2V102 ) 0    2.160000     3.420000
301      1398562443839      db2agntdp (DB2V102 ) 0    0.170000     0.190000
298      1398553643836      db2agent (DB2V102) 0     2.910000     5.110000
```

当发现用户 CPU 时间比例过高时，就需要分析占用 CPU 过高的 SQL 语句或者应用程序。通过使用应用程序快照，或查询 sysibmadm.snapappl 管理视图来找出什么应用程序消耗了大多数 CPU 时间，从下面的查询结果看出，主要是应用程序消耗了用户 CPU：

```
select appl.dbpartitionnum, appl_name,
        agent_usr_cpu_time_s + agent_usr_cpu_time_ms / 1000000.0 as user_cpu
 from sysibmadm.snapappl appl,
       sysibmadm.snapappl_info appl_info
 where appl.agent_id = appl_info.agent_id and
        appl.dbpartitionnum = appl_info.dbpartitionnum
order by user_cpu desc

DBPARTITIONNUM APPL_NAME                USER_CPU
-------------- ------------------------- ---------------------------------
             0 db2jcc_application                   3010.51562500000
             0 db2jcc_application                   2699.07812500000
             0 db2jcc_application                   2216.73437500000
             ……    ……                               ……
             0 Spotlight.exe                        204.10937500000
             0 db2jcc_application                    174.28125000000
             0 db2stmm                              34.04687500000
             0 db2jcc_application                     11.29687500000
             0 db2bp.exe                             3.71875000000
             0 QuestSQLOptimizer_DB                    0.15625000000
```

同样，通过动态语句快照，或查询 sysibmadm.snapdyn_sql 管理视图，可以确定哪些 SQL 语句正在占用绝大多数 CPU 时间：

```
select substr(stmt_text,1,200) as sql_text,
 total_usr_cpu_time +
 total_usr_cpu_time_ms / 1000000.0 as user_cpu
 from sysibmadm.snapdyn_sql
 order by user_cpu desc

sql_text                                                  user_cpu
----------------------------------------------------- …   -----------------
select  q1."ACTIVERECORD",  q1."AIRPORTCATEGORYCAF",     14644.09375000000
select  q1."ACTIVERECORD",  q1."SCHEDULEDAIRLINE",  q     10712.26562500000
select  q1."ACTIVERECORD",  q1."ARRIVALAIRPORT",  q1.      4304.67187500000
select  q1."ACTIVERECORD",  q1."ARRIVALAIRPORT",  q1.      3618.21875000000
......                                                    ......
```

利用下面的查询，可以查出占用 CPU 最高的前 10 个动态 SQL 语句：

```
SELECT  MEMBER,
        SECTION_TYPE,varchar(stmt_text,200) as statement,num_exec_with_metricsas
        numExec, TOTAL_CPU_TIME/NUM_EXEC_WITH_METRICS as
        AVG_CPU_TIME, TOTAL_CPU_TIME
```

```
FROM TABLE(MON_GET_PKG_CACHE_STMT( 'D', NULL, NULL, -2)) as T
WHERE T.NUM_EXEC_WITH_METRICS <> 0
ORDER BY AVG_CPU_TIME desc
FETCH FIRST 10 ROWS ONLY;

MEMBER SECTION_TYPE STATEMENT              NUMEXEC    AVG_CPU_TIME TOTAL_CPU_TIME
------ ------------ -------------------- -------------- ------------ ------------
     0 D            SELECT  MEMBER, SECTION…        2        26524          53048
     0 D            select * from syscat.tab…     33        26200.515     864617
     0 D            select * from act…             34        97.147058       3303
```

同时，还应该考虑静态 SQL 语句。快照监控器并没有包含静态 SQL 语句的信息。所以需要使用语句事件监控器，或者 WLM 活动事件监控器，来收集它们的系统 CPU 和用户 CPU 的使用信息。可以用 LIST APPLICATIONS 命令得到关注的应用程序 ID（例如 LOCAL.DB2.075D83033106），然后使用下面的语句定义监视器：

```
create event monitor stmt_evt for statements
 where appl_id = '*LOCAL.DB2.075D83033106'
 write to table
 connheader（table stmt_evt_ch, in tbs_evmon）,
 stmt（table stmt_evt_stmt, in tbs_evmon, trunc）,
 control（table stmt_evt_ctrl, in tbs_evmon）
 buffersize 512
```

通过以上事件监视器结果来进一步分析，通常对于 CPU 受限的环境中，用户 CPU 是决定因素，如果发现系统 CPU 时间过高，则需要查看 DB2 线程在操作系统中的上下文切换率。在 Unix 系统中，上下文切换在 vmstat 中在“CS”列有记录。每秒超过 75000~100000 次的上下文切换速度说明切换率过高。这时要减少用户连接数或者 DB2 代理程序数。

```
=> vmstat
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu--
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0   5488 550396  95596 872224    0    0    23    13   86  212  0  1 99  0  0
```

6.6.2 I/O 瓶颈的监控与诊断

当使用 IOSTAT 或者 TOPAS 等系统命令发现某个磁盘特别繁忙时（繁忙程度超过 80%），则说明存在磁盘瓶颈。确定存在瓶颈的磁盘在系统中的路径后，可以利用下面的查询得到哪个表空间用到了这个磁盘：

```
select substr(tbsp_name,1,25) tbsp_name, tbsp_id, substr(container_name,1,50)
       container_name
from sysibmadm.snapcontainer
where container_name like 'F:%

TBSP_NAME                 TBSP_ID              CONTAINER_NAME
------------------------- -------------------- -----------------------------
TSBCPD002                                    4 F:\db2data\TSBCPD002C001.DAT
TSBCPD002                                    4 F:\db2data\TSBCPD002C002.DAT
TSBCPD002                                    4 F:\db2data\TSBCPD002C003.DAT
```

当得到表空间的名字 **TSBCPD002** 和标号 4 后，根据表空间的属性，有以下的分析方法。

（1）如果这个表空间是用于存放表数据的，则通过下面的查询可以得到这个表空间中所有表的行读写次数，确定“热表”：

```
SELECT TABLE_NAME, ROWS_READ, ROW_WRITTEN
FROM sysibmadm.snaptab
WHERE TABLE_NAME IN (SELECT TABNAME FROM SYSCAT.TABLES WHERE
        TBSPACEID=4)

TABNAME                   ROWS_READ          ROWS_WRITTEN
------------------------- -------------------- --------------------
DATAINTEGRITYARCHIVE                    47                  94
DATAINTEGRITYMESSAGE             445132850              807779
FLIGHTDETAIL                       5696899             4229862
AIRPORTTRAFFIC                    84936644              250261
ARCHIVEDINPUT                      9701900             1844624
CHARGE                             8884354             9390068
DETERMINEDSERVICE                 12050486             9907543
FLIGHTDATA                        12442784             1594678
FLIGHTDETAILHISTORY             2262230518             1273347
INVOICELINEITEM                    8293936             2593130
MATCHRESULT                        2592199              664003
```

使用下面的查询可以查出所有涉及到“热表”**FLIGHTDETAILHISTORY** 的动态 SQL 语句：

```
SELECT substr(STMT_TEXT,1,50) as STMT_TEXT,ROWS_READ from sysibmadm.snapdyn_sql
 WHERE translate(cast(substr(stmt_text,1,50) as varchar (55)))
like 'EMPLOYEE' order by ROWS_READ

STMT_TEXT                                      ROWS_READ
------------------------------------------  -------- --------------------
SELECT * FROM FLIGHTDETAILHISTORY WHERE ….        1262230518
......
```

上面的查询返回表的行读写次数（ROWS_READ、ROWS_WRITTEN）、执行时间、执行次数（NUM_EXECUTIONS）、CPU 时间等等。使用 ROWS_READ、ROWS_WRITTEN 和 NUM_EXECUTIONS 对结果进行排序，找出最活跃的 SQL 语句。

除了动态语句，如果要判断是否是静态 SQL 语句造成了瓶颈，可以使用下面的语句找出所有访问热表 **FLIGHTDETAILHISTORY** 的静态 SQL：

```
select PKGSCHEMA, PKGNAME, SECTNO, substr（TEXT,1,80）
from syscat.statements
where translate（cast（substr（text,1,32672）
        as varchar（32672））） like ‘ %FLIGHTDETAILHISTORY% ’

PKGSCHEMA  PKGNAME       SECTNO 4
---------- ------------ ------ ------------------------------------------------------
NULLID    P1418126958        1 SELECT ARRAY_AGG(A1.LOCK_WAIT_START_TIME), ARRAY_AGG(A
NULLID    P1418126958       74 SELECT ARRAY_AGG(A1.AGENT_TID), ARRAY_AGG(A1.REQUEST_T
NULLID    P1418126958       85 SELECT ARRAY_AGG(A1.ACTIVITY_ID), ARRAY_AGG(A1.UOW_ID)
```

```
NULLID   P966173830       26 SELECT ARRAY_AGG(A.ACTIVITY_ID ORDER BY (A.CPU_RANK +
......
```

然后可以使用 db2pd -static 来找出它们当中哪些是高度活跃的。db2pd -static 的输出中，NumRef 列显示这条语句已经运行了多少次，UseCount 列显示了当前有多少 DB2 代理程序正在运行这条语句，找出 NumRef 次数最高行对应的 PkgName 值(P1418126958)和 SecNo 值(1)，就可以从上面的结果中查询到该 SQL 语句:

```
Sections:
Address            Schema   PkgName   UniqueID SecNo NumRef     UseCount
0x00007FBB449F72E0 NULLID   P1418126958         1    1662       0
0x00007FBB449F7530 NULLID   P1418126958         2    0          0
0x00007FBB449F7780 NULLID   P1418126958         3    831        1
0x00007FBB449F79D0 NULLID   P1418126958         4    0          0
0x00007FBB449F7C20 NULLID   P1418126958         5    830        0
0x00007FBB449F7E70 NULLID   P1418126958         6    0          0
0x00007FBB449F80C0 NULLID   P1418126958         7    831        1
0x00007FBB449F8310 NULLID   P1418126958         8    0          0
0x00007FBB449F8560 NULLID   P1418126958         9    830        0
0x00007FBB449F87B0 NULLID   P1418126958         10   830        0
0x00007FBB449F8A00 NULLID   P1418126958         11   0          0
```

（2）如果这个表空间是一个索引表空间，需要检查表空间中是否有活跃的索引读写，如下面的表空间 **TSBCPD003** 中有大量的索引读(POOL_INDEX_P_READS)：

```
 select dbpartitionnum, tbsp_id, tbsp_name,
        pool_index_p_reads, pool_index_writes
 from  sysibmadm.snaptbsp T
 where T.tbsp_id = 5

DBPARTITIONNUM TBSP_ID    TBSP_NAME        POOL_INDEX_P_READS   POOL_INDEX_WRITES
-------------- ---------- ---------------- -------------------- --------------------
             0          5 TSBCPD003       262230518            0
```

接着查看表空间 **TSBCPD003** 中有哪些索引及使用该索引的表：

```
 select t.tabname, i.indname
 from syscat.tables t, syscat.indexes i
 where t.tabname = i.tabname and
       coalesce(t.index_tbspace, t.tbspace) = 'TSBCPD003'

TABNAME                                                 INDNAME
------------------------------------------------------- -----------------------
FLIGHTDETAILHISTORY                                     SQL121023180300770
FLIGHTDETAILHISTORY                                     SQL121023180300860
FLIGHTDETAILHISTORY                                     SQL121023180300870
```

然后查看其中哪些索引是高度活跃的。如果在检查的表空间中有不止一个索引，就需要查看索引层面的活跃程度。反复运行命令 db2pd~db <dbname>-tcbstats index。其中“TCB index Stats”列显示了所有活动的索引，以及每一个的统计信息。“Scans”列显示了在每个索引上已经执行了多少次索引扫描。

```
db2pd -db FLIGHTDB  -tcbstats index
Database Member 0 -- Database FLIGHTDB -- Active -- Up 17 days 00:11:48 --

……

TCB Index Stats:
Address            TableName                    Scans      IxOnlyScns KeyUpdates
0x00007F9DF3F56A60 SYSTABLES                    58         58         0
0x00007F9DF3F56A60 SYSTABLES                    0          0          0
0x00007F9DF3F56A60 SYSTABLES                    0          0          0
0x00007F9DF3F56A60 SYSTABLES                    0          0          0
0x00007F9DF3F56A60 SYSTABLES                    0          0          0
0x00007F9DF3F759E0 FLIGHTDETAILHISTORY          129700     1600       0
......
```

（3）如果这个表空间是临时表空间，下面的查询可以查出这个临时表空间的 I/O 繁忙是否由于排序溢出(SORT_OVERFLOWS)导致的：

```
select dbpartitionnum, total_sorts,
       total_sort_time, sort_overflows
 from sysibmadm.snapdb

DBPARTITIONNUM   TOTAL_SORTS          TOTAL_SORT_TIME      SORT_OVERFLOWS
-------------- -------------------- -------------------- --------------------
             0            10638459              2850366               222797
```

或者通过以下查询可以查询中间结果数据的读写情况，从而确定 I/O 是否由于过多的中间结果导致的：

```
select dbpartitionnum, tbsp_name,
        pool_temp_data_p_reads, pool_data_writes
from sysibmadm.snaptbsp
where tbsp_id = 3

DBPARTITIONNUM TBSP_NAME     POOL_TEMP_DATA_P_READS POOL_DATA_WRITES
-------------- ------------- ---------------------- --------------------
             0 TEMPSPACE1                         0                75866
```

6.6.3　内存瓶颈的监控与诊断

DB2 的内存问题分为物理内存限制和内存的分配问题，现实中，通常物理内存是充足的，但是有时候也会表现为内存不足，这是因为默认参数的情况下，分配给 DB2 使用的内存太小造成的，所以对于只安装 DB2 的系统来说，通常应该将 60%以上的内存分配给 DB2。另一个问题是内存在 DB2 内部不同组件的分配，自从 DB2 V9 引入强大的 STMM 机制之后，DBA 对于内存管理的工作大大减轻了。但是正确分配充足的内存仍然是良好性能的关键，而且一个内存瓶颈经常能够把自己伪装成磁盘或 CPU 问题，所以在发现磁盘或者 CPU 问题的时候通常需要进一步分析，以找到真正的罪魁祸首。表 6-10 列出了常见的内存瓶颈，但却表现为磁盘或者 CPU 的问题。

表 6-10　内存问题导致的磁盘 CPU 问题

内存问题	症状
缓冲太小或者整个系统内存太小	数据或者索引表空间的磁盘瓶颈
Sortheap/sheapthres_shr 太小	临时表空间磁盘瓶颈
日志缓冲太小	日志磁盘瓶颈
包高速缓冲太小	重复包缓冲插入造成的 CPU 瓶颈
系统内存过度分配或者使用较小的内存页大小管理很大的系统内存	过多的系统 CPU 时间花费在虚拟内存管理（VMM）上

当出现 CPU 或者磁盘问题的时候，需要查看数据库、实例和应用的内存使用状态，以确定是否由于内存的原因导致。这可以通过快照、管理视图或者 db2pd 来查看。

下面介绍如何使用 db2pd 命令进行查看，使用下面的命令可以从 Size 和 Used 列查看数据库分配和使用的总内存：

```
[db2inst1@db2 ~]$db2pd -memsets -db FLIGHTDB

Database Partition 0 -- Database FLIGHTDB -- Active -- Up 2 days 23:35:50

Memory Sets:
Name ...  Id         Size(Kb)  ... Used(Kb)   Cmt(Kb)    Uncmt(Kb)
FLIGHTDB     0          57905792     35188480   35851392   128
App265    0          0           0          0          0
App1348   0          0           0          0          0
App701    104690      128          128        128        0
```

使用下面的命令可以查看各种堆使用的实际内存，通过分析请求内存的大小与实际分配的物理内存情况来确定是否存在内存问题以及存在内存问题的组件，然后结合第 7 章的参数调整来调整数据库参数，将数据库内存分配到最佳状态。

```
[db2inst1@db2 ~]$db2pd -mempools -db FLIGHTDB

Database Partition 0 -- Database FLIGHTDB -- Active -- Up 2 days 23:36:10

Memory Pools:
... MemSet   PoolName   Id ... LogSz  ...  PhySz      PhyHWM   ... CfgParm
    FLIGHTDB    utilh      5      4928        458752      524288      UTIL_HEAP_SZ
    FLIGHTDB    pckcacheh  7      39145186    42074112    51642368    PCKCACHESZ
    FLIGHTDB    xmlcacheh  93     149656      196608      196608      n/a
    FLIGHTDB    catcacheh  8      1036888     1376256     1441792 CATALOGCACHE_SZ
    FLIGHTDB    bph        16    1070633344  1071185920  1071185920  n/a
    FLIGHTDB    bph        16    1070633344  1071185920  1071185920  n/a
    FLIGHTDB    bph        16    1070633344  1071185920  1071185920  n/a
    FLIGHTDB    bph        16    1070633344  1071185920  1071185920  n/a
    ......
```

使用下面的命令可以查看缓冲池状态，得到逻辑读、物理读以及命中率等信息，结合前面所讲解的 KPI 就能够确认是否存在由于缓冲池造成的内存瓶颈：

```
[db2inst1@db2 ~]$db2pd -bufferpools -db FLIGHTDB

Database Partition 0 -- Database CASMS -- Active -- Up 2 days 23:36:29

Bufferpools:
First Active Pool ID      1
Max Bufferpool ID        11
Max Bufferpool ID on Disk 11
Num Bufferpools          15

......
Bufferpool Statistics for all bufferpools (when BUFFERPOOL monitor switch is ON):

BPID DatLRds    DatPRds    HitRatio ... HitRatio ... HitRatio TmpIdxLRds HitRatio
1    492071     758        99.85%      100.00%       99.88%   0          00.00%
2    34         34        00.00%      00.00%       00.00%   0          00.00%
3    2584949464 75174005   97.09%      00.00%       00.00%   0          00.00%
4    234174401  5730      100.00%      00.00%       00.00%   0          00.00%
5    16597275   462       100.00%      00.00%       99.92%   0          00.00%
...
```

6.6.4 懒惰系统的监控与诊断

有一种性能问题尽管没有明显的磁盘瓶颈、内存瓶颈、CPU 瓶颈或网络瓶颈，但是系统的响应就是出奇的慢，这就是通常所说的懒惰系统。懒惰系统通常和锁相关，包括锁等待、锁升级、死锁和锁超时。

可以通过下面命令确定系统是否存在严重的锁等待。

```
select snap.appl_name, snap.primary_auth_id,
      ap.lock_waits, ap.lock_wait_time
 from sysibmadm.snapappl_info snap,sysibmadm.snapappl ap
 where snap.agent_id = ap.agent_id and ap.lock_waits > 0

APPL_NAME            PRIMARY_AUTH_ID     LOCK_WAITS           LOCK_WAIT_TIME
-------------------- ------------------- -------------------- --------------------
......
db2jcc_application   DB2ADMIN                           367              322325
db2jcc_application   DB2ADMIN                           281              158276
db2jcc_application   DB2ADMIN                           388              393523
db2jcc_application   DB2ADMIN                           512              441493
db2jcc_application   DB2ADMIN                           521              932273
db2jcc_application   DB2ADMIN                           954              739977
db2jcc_application   DB2ADMIN                          1169              842721
。......
```

通过如下所示的语句来监控锁升级、死锁和锁超时，通常每 1000 笔事务发生死锁超过一次，就意味着有问题。

```
select snap.appl_name, snap.primary_auth_id,
    ap.locks_held,ap.lock_escals, ap.lock_timeouts,
    ap.deadlocks,ap.int_deadlock_rollbacks
 from sysibmadm.snapappl_info snap,sysibmadm.snapappl ap
```

```
     where snap.agent_id = ap.agent_id

    APPL_NAME         ......  LOCKS_HELD   LOCK_ESCALS  LOCK_TIMEOUTS  DEADLOCKS
INT_DEADLOCK_ROLLBACKS
    ----------------------    ------------ ------------- -------------- ----------
----------------------
    ......
    db2jcc_application         0          0          0            42          24
    db2jcc_application         0          0          0            59          25
    db2jcc_application         0          0          0            66          29
    db2jcc_application         0          0          0            89          42
    db2jcc_application         0          0          0            96          50
```

可以通过以下方法来避免：

（1）如果可能，尽可能避免运行时间长的事务和 WITH HOLD 的游标。持有锁的时间越长，与其他应用程序发生锁争抢的可能性就越大。

（2）尽可能只查询实际需要的最小结果集，尤其是在 REPEATABLE READ 隔离级别时。因为涉及的行数越多，发生锁争用的可能性就越大。

（3）避免使用过高的隔离级别，隔离级别越高，发生锁问题的可能性就越大。

（4）如果应用程序的商业逻辑允许的话，可以通过 DB2 V9.5 提供的这三个注册变量 db2_evaluncommitted、db2_skipdeleted 和 db2_skipinserted 来更改锁的行为。这些设置可以让 DB2 推迟或者避免在某些情况下产生锁，因而减少潜在的锁争用。

（5）如果所有的应用以相同的顺序来访问表中的数据，这样可以降低死锁的概率。

（6）如果是使用 DB2 V9.7 及以上的版本，将 CUR_COMMIT 设置为 ON，通过其引入的写不阻塞读特性，能够大大降低锁引发的问题。

6.7　图形化性能管理器

上面介绍了各种命令行工具来分析和解决性能相关的问题，但是这对 DBA 的知识技能要求较高，而且对于解决复杂的问题，即使是比较资深的 DBA 也需要花费大量的时间，效率较低。而图形化的性能管理器能够使 DBA 更容易的发现问题，并且能够在相关的信息之间快速导航，从而加速问题的分析与诊断，并快速地解决问题。

Optim Performance Expert Manager(简称 OPM)作为 IBM 原厂提供的图形化工具，通过将各种监控信息直观的表现出来，使 DBA 易于发现问题。

根据不同系统的特点，OPM 提供了设置监控的模板，这些模板针对不同系统的特点打开不同的监控开关从而获取相应的监控信息。还针对重要的信息设置相应的阀值，当达到或超过这些阀值时，OPM 能够灵敏地将捕获地数据库信息以警报形式迅速发布。

OPM 通过图表的形式直接展示系统运行情况，通过图形界面钻取详细信息，直至揭示问题的根源。

OPM 提供的性能数据资源库，能够保存监控信息的历史，通过观察历史信息发展的趋势，

可以提前避免问题，帮助数据库管理员“防患于未然”，是解决 DB2 性能问题的强有力工具。

由第三方公司提供的 Toad for DB2 是另外一个比较实用的工具，它的 Spotlight on DB2 模块通过一个简单的图形化界面实时显示 DB2 所有的活动状态。它的结构化视图精确地展现了整个 DB2 环境，通过它们可以看到 DB2 的活动和健康情况，包括连接、SQL、等待事件、锁、内存和磁盘 I/O 等等相关信息，并在关联的信息之间快速导航，从而能够帮助指出问题的本质原因并快速地找到解决方案。

6.8 小结

本章首先介绍了系统监控的方法和工具使用，包括 CPU、I/O 和内存的监控；随后对数据库常用监控手段作了详细介绍；另外，为了帮助 DBA 掌握性能监控标准，给出了常用的数据库关键性能指标（KPI）；最后，从性能优化的角度，向广大 DBA 分享了 DB2 监控的最佳实践。

第 7 章 配置参数与运维工具优化

现在人们越来越讲究生活品质，而生活的品质包括方方面面，首先我们必须有一个健康的身体，通过体检，能够监控各项生理指标，当超出正常范围的时候，需要通过医治或者锻炼来使其达到正常；其次我们需要一个良好的环境，当环境状态出现问题的时候，需要通过绿化等手段来改善环境。

DB2 参数就像人体各项生理指标，也需要进行调整优化才能良好地运行，而操作系统作为 DB2 的运行环境，也必须优化其参数，然后通过各种运维工具的使用，以使数据库系统发挥其最大的价值。

7.1 配置参数优化

运动员必须根据所从事的项目不同，监控和调整相应的生理指标，以使其在大赛前达到最合理的数值，从而获得好的比赛成绩。对于 DB2 来说，其配置参数就像人体生理指标，必须根据生产系统的特征，不断地监控和调整，以达到最合理的数值。但是 DB2 中可以调节的参数多种多样，并且参数之间的相互关系错综复杂，很容易牵一发而动全身。如果忽视对参数改动后的副作用，那么优化参数的行动就犹如“扫尘，一面扫，一面生”，解决了一个老问题，还会产生新问题，甚至是严重的新问题。

设置参数实际上是在分配资源，需要了解每种资源的特点和数据库中消耗资源的各个对象之间的依赖关系，依据具体情况，对参数设定和调整。

调优参数时需要注意如下三点：

（1）要对工作负载和系统资源有深刻的理解，应该根据最典型的工作负载状况设置参数。每个系统都有它独特的软硬件配置和工作负载，所以参数配置也应该量身定制，不偏不倚，切忌把理论和经验生搬硬套。

（2）多参考现有系统的配置。如果已有运行良好、业务特征相似的系统，则可以依据这些系统中的相关配置，对新系统进行配置，因为它们已被证明是可靠的。

（3）可以考虑先设置初始值，然后进行微调。微调时，每次只调整一个参数，并保留调整记录。如果调整无效，重设回来。切忌盲动，要有的放矢、谋定而后动、按部就班地进行参数优化。

配置和优化系统参数的工作顺利完成后，CPU、内存、磁盘和网络的处理能力会得到充分的利用，并高效地运转。本节将从操作系统层次开始，讨论与性能相关的参数配置和优化方法。

7.1.1 操作系统参数优化

调整操作系统参数，以满足 DB2 数据库的要求，是 DB2 高效运行的基础。DB2 可以运行在多种操作系统上，下面介绍如何对几种常用操作系统进行参数优化。

1. AIX 配置

如果 DB2 运行在 AIX 系统上，可以利用 VMO 命令（该命令用于调整虚拟内存系统的各种参数）将系统参数 lru_file_repage 设为 0，minperm 设为 3。这将使 DB2 从操作系统直接获取数据文件页面。这两个参数值是 AIX 版本 6.1 的默认值。可以使用如下命令设置（其中≠p 表示立即生效，而≠o 表示设置并显示参数值）：

```
vmo -p -o lru_file_repage=0
vmo -p -o minperm%=3
```

此外应根据需要调整系统参数 maxservers。这个参数是处理异步 I/O 的内核进程数，默认为每个 CPU 内核 10 个。可以根据系统压力适当增大。

参数 maxreqs 应该被设置成 MAX（NUM_IOCLEANERS×256,4096）。这个参数控制绝大

多数突发的异步 I/O 请求。通过如下命令将 maxservers 设置为 100，maxreqs 设置为 4096（其中≠l 指定设备的逻辑名称，≠a 指定要更改的属性）：

```
chdev -l aio0 -a maxservers=100
chdev -l aio0 -a maxreqs=4096
```

磁盘参数 queue_depth 应该基于队列的物理硬盘数目设置。例如，对于 IBM 磁盘 queue_depth 的默认值是 3，建议这个值设为（3×磁盘数目）。这个参数控制可排队的磁盘请求。如下命令设置 queue_depth 为 18：

```
chdev -l hdisk20 -a queue_depth=18
```

磁盘适配器参数 num_cmd_elems 控制排队发送到适配器的请求数，应该设置为所有与适配器相连接的设备 queue_depth 之和，如下命令设置 num_cmd_elems 值为 80：

```
chdev -l SCSI2 -a num_cmd_elems=802. Solaris 和 HP-UX 配置
```

对于在 Solaris 或 HP-UX 平台，可以利用 DB2 专门提供的内核参数设定工具 db2osconf 得到内核参数推荐值。db2osconf 基于内存和 CPU 能力，以及缩放系数（期望的工作负载与当前工作负载的比率）来进行配置。运行大型系统如 SAP 应用程序时，应该使用 2 或者更高的缩放系数。在 Solaris 和 HP-UX 平台，db2osconf 提供了很好的初始值。

下面是对于一个需要 500 个并发线程的 DB2 运行 db2osconf 的输出结果（−t 指定并发线程数）：

```
=>db2osconf -t 500

set msgsys:msginfo_msgmax = 65535
set msgsys:msginfo_msgmnb = 65535
set msgsys:msginfo_msgssz = 32
set msgsys:msginfo_msgseg = 32767
set msgsys:msginfo_msgmap = 2562
set msgsys:msginfo_msgmni = 2560
set msgsys:msginfo_msgtql = 2560
set semsys:seminfo_semmap = 3074
set semsys:seminfo_semmni = 3072
set semsys:seminfo_semmns = 6452
set semsys:seminfo_semmnu = 3072
set semsys:seminfo_semume = 600
set shmsys:shminfo_shmmax = 2134020096
set shmsys:shminfo_shmmni = 3072
set shmsys:shminfo_shmseg = 600

Total kernel space for IPC:
0.35MB (shm) + 1.77MB (sem) + 1.34MB (msg) == 3.46MB (total)
```

获得建议值之后，对于 Solaris 系统通过在 /etc/system 文件末尾添加参数值来修改。将以下行添加到/etc/system 文件的末尾，然后重启系统，可以将 msginfo_msgmax 值更改 65535。

```
set msgsys:msginfo_msgmax = 65535
```

对于 HP-UX 可以通过“系统管理器”（SAM） 程序来修改，更改内核配置参数的值之后，

HP-UX 操作系统将自动重新启动以应用新值。

3. Linux 配置

在 Linux 系统上，SHMMAX（共享内存的最大值）应设为可用内存的 90%以上。对于 16G 内存的计算机，其进程间通信内核配置参数以及推荐值为：

```
kernel.shmmni=4096  #（共享内存段的最大数目）
kernel.shmmax=17179869184  #（共享内存段的最大字节数）
kernel.shmall=8388608  #（总的共享内存大小，单位：字节）
#kernel.sem=<SEMMSL> <SEMMNS> <SEMOPM> <SEMMNI>
kernel.sem=250  256000  32  4096 #（内核旗语）
#semmsl 表示一个信号量集合中能够包含的信号量最大数目。
#semmns 表示系统内可允许的信号量最大数目。
#semopm 表示单个 semopm()调用在一个信号量集合上可以执行的操作数量。
#semmn 表示系统信号量集合总数。
kernel.msgmni=16384 #（消息队列标识的最大数目）
kernel.msgmax=65536 #（一个消息的最大大小，单位：字节）
kernel.msgmnb=65536 #（消息队列的默认大小，单位：字节）
```

如果实际环境中物理内存多于 16GB，下列参数值 SEMMSL、SEMMNS、SEMOPM、MSGMAX 和 MSGMNB 可以保持不变，但是如下参数值 SHMMNI、SHMMAX、SHMALL、SEMMNI 和 MSGMNI 应成比例增大。例如对于 64G 内存的机器，需要将 SHMMAX 设置成 **17179869184×4= 68719476736。**

对于以上内核参数可以通过编辑 /etc/sysctl.conf 文件来修改内核参数值。如果此文件不存在，那么应创建它。修改后通过执行“sysctl －p”以使参数生效。

7.1.2　DBM 参数优化

数据库管理器（DBM）参数也称为实例参数，因为这些参数都是实例级别的，对该实例下所有数据库的性能都有影响。可以通过如下命令查看所有的 DBM 参数值，其中 show detail 表示显示参数的详细信息（例如延迟生效参数的更新值）：

```
db2 get dbm cfg show detail
```

下面介绍重要的 DBM 参数及其优化方法。

1. MAXAGENTS

该参数指定 DB2 中处理应用程序请求的代理程序的最大数目。MAXAGENTS 的值应当大于同时活跃的数据库的 MAXAPPLS（并发应用程序最大数目）值的总和。最安全的方案是将这个参数值设为 NUMDB（最大活动数据库数）和 MAXAPPLS 的乘积。在通常情况下，将 MAXAGENTS 的值设置成略微大于数据库并发连接的最大预计数。

数据库管理器的快照信息中，“Agents waiting for a token”或“Agents stolen from another application”跟 MAXAGENTS 的设定有关。在数据库运行期间，可以使用下面这个命令来获取数据库管理器的快照数据：

```
db2 get snapshot for database manager
```

示例输出如下：

```
High water mark for agents registered = 4
High water mark for agents waiting for a token = 0
Agents registered = 4
Agents waiting for a token = 3
Idle agents = 0
Agents assigned from pool = 5
Agents created from empty pool = 4
Agents stolen from another application = 5
High water mark for coordinating agents = 4
Max agents overflow = 0
```

如果发现“Agents waiting for a token”或“Agents stolen from another application”不等于 0，则需要增加 MAXAGENTS，以允许数据库管理器可以创建和使用更多的代理程序。通过如下命令可以设置 MAXAGENTS 参数值为 300：

```
db2 update dbm cfg using MAXAGENTS 300
```

2. NUM_POOLAGENTS

该参数用于控制代理程序池的大小。DB2 接到一个新的请求时，会从代理程序池中取出一个空闲的代理程序用于处理请求。如果代理程序池中没有空闲的代理程序，则会临时创建一个。如果代理程序数目多于 NUM_POOLAGENTS，那么当代理程序执行完请求后，将终止运行，而不是返回到代理程序池。

在存在大量并发请求的 OLTP 环境中，为避免 DB2 频繁创建和终止代理程序，应将 NUM_POOLAGENTS 设置为接近于或等于 MAXAGENTS 的值。可以通过如下命令调整将该参数调整为 300，使用其等于 MAXAGENTS 参数值：

```
db2 update dbm cfg using NUM_POOLAGENTS 300
```

3. NUM_INITAGENTS

该参数指定数据库启动阶段在代理程序池中创建的代理程序的数量。在系统刚启动并开始接受用户连接时，这些提前创建的代理可以直接用于处理用户请求。在大多数情况下，让 NUM_INITAGENTS 保留为缺省值会比较好。

4. FCM_NUM_BUFFERS

这个参数只有在 DPF 环境下才有用，它指定用于内部通信的大小为 4KB 的缓冲区数量。

5. INTRA_PARALLEL

此参数指定数据库管理器是否可以使用分区内并行性。对于 OLTP 系统，其并发连接数通常较多，这时关闭分区并发性较好，对于 OLAP 等并发连接较少并且 SQL 较复杂的情况下，开启分区并行性通常能够获益。对于混合工作负载通常不启用分区内并行性。

6. MON_HEAP_SZ

此参数确定分配给数据库系统监视器的内存量（以页计）。当进行数据库监视活动时，从

监视器堆分配内存。如果没有足够的可用内存，并且 DB2 返回如下错误号（ADM2004E、ADM2005W、ADM2006W）之一时，可以将值设为 1024。如果还是遇到错误，可以每次增加 256 直到错误消失。

7. QUERY_HEAP_SZ

此参数是为了将每个查询存储到代理的私有内存时可以分配的最大内存量。可以用如下命令调整为 250（4k 页大小）。

```
db2 update dbm cfg using QUERY_HEAP_SZ 250
```

7.1.3 DB 参数优化

DB 参数也就是数据库配置参数，可以使用数据库配置建议程序，也就是通常所说的 AUTOCONFIGURE 命令进行设定。这个工具根据提供的系统负载情况，自动生成一系列数据库配置参数的初始值。使用 AUTOCONFIGURE 应该注意下面几点：

- 在数据库完成数据装入后，再运行 AUTOCONFIGURE。这样做的原因是，让 AUTOCONFIGURE 计算推荐值时，依据正常状态下的数据量，太多或太少的数据都将降低计算的精度。
- 对所有选项指定符合实际的值，因为默认值可能并不适用于你的系统。例如 MEM_PERCENT 默认值设置为 25%，对于一个只运行 DB2 数据库的服务器来说这个值太低了，应该设为 60%以上。
- 使用“APPLY NONE”选项反复运行 AUTOCONFIGURE 来试验不同的选项值，并对比所得到的结果。

下面是一个运行 autoconfigure 工具的例子：

```
    DB2 AUTOCONFIGURE USING MEM_PERCENT 60 WORKLOAD_TYPE MIXED NUM_STMTS 20 TPM 500
ADMIN_PRIORITY BOTHIS_POPULATED YES NUM_LOCAL_APPS 0 NUM_REMOTE_APPS 20 ISOLATION
RRBP_RESIZEABLE YES APPLY NONE
```

autoconfigure 的参数及其含义如表 7-1 所示:

表 7-1　autoconfigure 命令的参数及有效值

参数名称	有效值	默认值	说明
mem_percent	1–100	25	分配给数据库的实例内存比例
workload_type	simple，mixed，complex	mixed	简单工作负载通常指 I/O 密集工作负载，而复杂工作负载通常是 CPU 密集负载
num_stmts	1–1 000 000	10	每工作单元包含的语句数目
Tpm	1–200 000	60	每分钟事务
Admin_priority	performance，recovery，both	both	性能优先还是恢复优先
is_populated	yes，no	yes	数据是否已经装载到数据库
num_local_apps	0–5 000	0	连接到数据库的本地应用数目

（续表）

参数名称	有效值	默认值	说明
num_remote_apps	0–5 000	10	连接到数据库的远程应用数目
isolation	RR，RSCS，UR	RR	连接到数据库应用的最高隔离级别
bp_resizeable	yes, no	yes	缓冲池大小是否能够调整
APPLY	DB ONLY，DB AND DB，NONE	无	指定命令作用的范围，当使用 NONE 时，只显示建议参数而不实际应用

AUTOCONFIGURE 一般会对内存相关的参数设置为 AUTOMATIC，这样就可以利用内存自动调节（STMM）功能进行自动管理。

AUTOCONFIGURE 通常用来进行初始设置，然后，DBA 还需要根据系统的运行情况进行持续的调整。下面就来探讨一下如何调整一些重要的数据库配置参数。

1. 缓冲池

数据库被创建时，会创建一个缺省的缓冲池，即 IBMDEFAULTBP。缓冲池的大小首先取决于 SYSCAT.BUFFERPOOLS 目录表中对应该缓冲池的 NPAGES 值。如果 NPAGES 为－1，则缓冲池的大小取决于数据库配置参数 BUFFPAGE。BUFFPAGE 参数的缺省值非常小，应根据数据库的大小和应用程序的性质，将该参数设置成一个合理的值。请考虑下列情况：

（1）对于大型 BI/OLAP 数据库，将 BUFFPAGE 设置成机器总内存的 50%。

（2）对于大型 OLTP 数据库，将 BUFFPAGE 设置成机器总内存的 75%。

通过如下命令可以将缓冲池 DATABP4K（PAGESIZE 为 4K）调整为 48G，并使用 immediate 选项使其立即生效：

```
db2 alter bufferpool DATABP4K immediate size -1
db2 update db cfg buffpage using 12000000
```

当然也可以通过如下命令直接将缓冲池大小设置为 48G：

```
db2 alter bufferpool DATABP4K immediate size 12000000
```

评价缓冲池最重要的指标是缓冲池命中率，利用快照或者管理视图可以得到缓冲池的命中率。下面的语句是利用 SYSIBMADM.BP_HITRATIO 得到命中率：

```
SELECT SUBSTR(db_name,1,8) AS DB_NAME, SUBSTR(bp_name,1,14) AS BP_NAME,
       total_hit_ratio_percent as TOTAL_HR, data_hit_ratio_percent as DATA_HR,
       Index_hit_ratio_percent as INDEX_HR, xda_hit_ratio_percent as XDA_HR,
       Dbpartitionnum as DBPARTNUM
FROM SYSIBMADM.BP_HITRATIO
ORDER BY dbpartitionnum
```

理想情况下，命中率应当超过 95%，并尽可能接近 100%。要提高缓冲池命中率，请尝试下面这些方法：

（1）增加缓冲池大小。

（2）考虑分配多个缓冲池。如果可能的话，为热表所属的表空间分配一个单独的缓冲池；

为冷表分配另外一个缓冲池。然后尝试改变这些缓冲池的大小，最终找到一个最合理的设置。

> **注意：缓冲池参数设置**
>
> 有时候不断增加缓冲池的大小并不能提高缓冲池命中率。这时问题的根源可能不在缓冲池的大小，继续增加缓冲池的大小可能事得其反，当缓冲池过大导致操作系统物理内存不足时，频繁使用虚拟内存反而会降低性能。

2. 私有排序阈值（sheapthres）、共享排序阈值（SHEAPTHRES_SHR）及排序堆大小（SORTHEAP）

sheapthres 是个实例级参数，表示整个实例范围内私有排序总内存的软限制，如果私有排序内存使用超过这个限制，则新请求的私有排序内存分配将会大幅减少。如果 sheapthres 被设置为 0 值，则 DB2 使用共享排序。如果 sheapthres 被设置为非 0 值，表示 DB2 将使用私有排序，排序内存在代理中分配，SORTHEAP 参数此时则规定了每个私有排序最多使用的内存量，IBM 信息中心的推荐是将 sheapthres 设置为 SORTHEAP 的整数倍，至少为实例中最大 SORTHEAP 值的 2 倍。

SHEAPTHRES_SHR 与 sheapthres 很容易混淆，为了便于大家理解，特列出它们二者之间的主要区别：

（1）SHEAPTHRES_SHR 中的 SHR 代表 Share，表示共享内存的限制，而 sheapthres 代表私有内存的限制。

（2）SHEAPTHRES_SHR 是数据库级参数，代表数据库范围的共享排序内存限制，而 sheapthres 是实例级参数，代表实例范围的私有排序内存限制。

（3）SHEAPTHRES_SHR 与 sheapthres 是冤家，如果 sheapthres 为 0，DB2 使用共享排序，SHEAPTHRES_SHR 才起作用，如果 sheapthres 不为 0，SHEAPTHRES_SHR 基本就没用了，除非开启了内部并行度（intra_parallel 为 YES）或者 max_connections 参数的值大于 max_coordagent，这时会使用共享排序。

（4）SHEAPTHRES_SHR 与 sheapthres 都是可以在线调整的，但 sheapthres 在从 0 调到非 0 值时需要重新启动实例才能生效。

要更改 SHEAPTHRES 和 SORTHEAP 的值，可以运行以下命令，其中将 SHEAPTHRES 设置为 0，表示启用共享排序，SORTHEAP 大小设置为 25600×4k=100M。

```
db2 update dbm cfg using SHEAPTHRES 0
db2 update db cfg for DB_NAME using SORTHEAP 25600
```

3. I/O 服务器的数目（NUM_IOSERVERS）

DB2 中访问表数据以及备份和恢复之类的操作都要利用 I/O 服务器（实际上是执行专门任务的代理程序）执行预取操作。当配置足够的 I/O 服务器时，可以大大提高能够受益于预取的查询的性能。为了最大程度地提高并行 I/O 的机会，NUM_IOSERVERS 至少需要等于数据库中的物理磁盘数。建议将此参数设为 AUTOMATIC，让 DB2 自动调节。

4. 异步页清除程序的数目（NUM_IOCLEANERS）

NUM_IOCLEANERS 指定了数据库的异步页清除代理程序的数目。这些页清除程序将缓冲

池中已更改的页异步地写到磁盘，从而释放这些页。这使得请求新页的代理程序不必等待已更改页被写到磁盘，就可以获得可用的新页，从而加快查询的处理。

如果将该参数设置成 0，则不启动页清除程序。这样 DB2 代理程序在使用缓冲池的时候，当发现没有立即可用的页时，必须要等待将缓冲池中已更改的页同步写到磁盘。另外，不启动页面清除，会使 DB2 无法充分利用磁盘的带宽，可能使某些磁盘有时处于不合理的空闲状态。

如果应用程序主要执行更新数据的事务，增加页清除程序的数目会提高性能。增加页清除程序的数量还会减少故障（比如断电）的恢复时间，因为大部分更新后的数据可能已经被写入磁盘而无须恢复。该参数也可以设为 AUTOMATIC，让 DB2 自动调节。

5. 更改页阈值（CHNGPGS_THRESH）

这个参数指定缓冲池中被更改页面占所有页面的百分比达到多少时，将启动异步页面清除程序，来将缓冲池中被更改的数据页异步写到磁盘。在只读环境下，不需要启动异步页面清除程序。在 OLAP 环境中，如果没有很多的 INSERT 或 UPDATE 语句，使用缺省值 60 就可以了。在 OLTP 环境中，一般将它设为 20~40 之间的一个值。这个值越低，页面清除程序将脏数据从缓冲池页面写到磁盘就越频繁，但是每次写的数据量会越少。

参看缓冲池相关的快照信息，如果“Dirty page steal cleaner triggers”是一个两位数，则试着降低这个参数。如果“Buffer pool data writes”较高，而“Asynchronous pool data page writes”较低，则也应该降低这个参数。

如果希望参数 CHNGPGS_THRESH 不起作用，可以设定注册变量 DB2_USER_ALTERNATE_PAGE_CLEANING 为 YES，这时将启用新的清页算法。使用新算法时，要把 NUM_IOSERVERS 至少设为 3，否则新算法的性能将受到影响。

6. MAXAPPLS

该参数指定了允许连接到 DB2 的并发应用程序的最大数量。由于需要给连接到 DB2 的每个应用程序分配私有内存，所以并发连接越多，内存消耗也会更多。

运行 OLTP 应用程序时，要保证 MAXAPPLS 的值能容纳最多的并发用户和连接。对于那些使用连接池的应用程序，建议将 MAXAPPLS 的值设置成比连接池的大小大 1 或 2，这样做只是为了以防需要调用命令行连接来同时做一些事情。

7. PCKCACHESZ

该参数指定包缓存的大小，包缓存是在数据库共享内存之外分配的，并且用于高速缓存数据库上的静态和动态 SQL 和 XQuery 语句的访问计划，在与数据库连接的应用程序多次使用同一个语句时，能够提高性能，这对于事务处理环境尤其重要。

8. CATALOGCACHE_SZ

该参数指定目录高速缓存可以使用的数据库堆中的最大空间（以页计）。目录高速缓存用于高速缓存系统目录信息。在分区数据库系统中，每个数据库分区都有一个目录高速缓存。高速缓存各个数据库分区中的目录信息允许数据库管理器不需要访问系统目录（或分区数据库环境中的目录节点）来获取先前检测的信息，从而降低其内部开销。使用目录高速缓存可以帮助

提高下列各项的整体性能：

- 绑定程序包以及编译 SQL 和 XQuery 语句。
- 涉及到检查数据库级别特权、例程特权、全局变量特权和角色权限的操作。
- 连接至分区数据库环境中的非目录节点的应用程序。

如果工作负载需要在短时间内编译许多 SQL 或 XQuery，且其后很少或不进行编译，那么增大此参数尤其重要。但是如果高速缓存太大，那么可能会因保留不再使用的信息的副本而浪费内存。在分区数据库的环境上，通常需要将目录节点上的 catalogcache_sz 设置为更大一些，因为非目录节点上需要的目录信息总是要先在目录节点上缓存。

9. DFT_QUERYOPT

在编译 SQL 和 XQuery 查询时，这个参数用于指导优化器使用不同程度的优化级，默认值是 5。在既没有使用 SET CURRENT QUERY OPTIMIZATION 语句，也没有使用绑定命令的 QUERYOPT 选项时，DB2 使用 DFT_QUERYOPT 这个参数的设置作为默认优化级。

10. 锁相关参数（LOCKLIST、MAXLOCKS 和 LOCKTIMEOUT）

与锁相关的 DB 参数包括 LOCKLIST、MAXLOCKS 和 LOCKTIMEOUT。

（1）LOCKLIST 设定锁列表缓冲区的最大空间。

（2）MAXLOCKS 指定了允许应用程序占用的锁列表的百分比。当锁列表耗尽或者一个应用程序所拥有的锁大于 MAXLOCKS×LOCKLIST 的时候，DB2 就会试图把几个行锁合并为一个表锁，从而释放锁空间。虽然锁升级本身并不耗费多少时间，但是锁住整个表通常会大大地降低并发性能。

（3）LOCKTIMEOUT 指定应用程序为获取锁所等待的最多时间，单位为秒。

- 如果将该参数设置成 0，那么应用程序将不会等待。如果锁请求没有得到满足，会返回 911 错误代码。
- 如果将该参数设置成-1，那么将关闭锁超时检测。在这种情形中，应用程序将一直等待到获得锁为止。

可以通过如下命令更改上面介绍的三个参数：

```
db2 update db cfg for sample using LOCKLIST 800
db2 update db cfg for sample using MAXLOCKS 50
db2 update db cfg for sample using LOCKTIMOUT 30
```

11. 应用程序堆大小（APPLHEAPSZ）

这个参数定义了一个代理程序或子代理程序可以使用的私有内存页数。在 DB2 初始化代理程序或子代理程序时，会先分配一个较小的堆空间。当代理程序或子代理程序需要更多的堆空间以处理较大的 SQL 语句时，数据库管理器将按照需要增加，直到所分配的内存大小达到该参数所指定的最大值。当应用程序接收到一个表明应用程序堆空间不够的错误号 SQL0954C 时，应该增加 APPLHEAPSZ 的值。

12. 日志相关参数

（1）LOGPRIMARY：指定要预先分配空间的主日志文件个数。

（2）LOGSECOND：当主日志文件个数不够用时，请启用从日志文件，从日志文件的最大个数就是这个参数的设定值。

（3）LOGFILSZ：日志文件的大小。

（4）LOGPATH/NEWLOGPATH：这个参数决定事务日志的位置。需要注意的是，事务日志不应和数据表空间共享磁盘，应放在有足够 I/O 能力的独立磁盘上。

（5）LOGBUFSZ：这个参数决定事务日志的内部缓存大小（单位为 4KB），默认值为 8，这个值应该设置更大一些。

（6）DIAGPATH：这个参数决定了 DB2 诊断日志文件的存放位置。在 DPF 环境中，应该将 DIAGPATH 设置为每个分区的本地非 NFS 路径。这样可以避免所有分区向同一个文件写入诊断信息。

（7）MINCOMMIT：一直等到事务请求数达到这个值，日志才会写到磁盘，这将有助于减少日志文件 I/O 的次数。MINCOMMIT 的缺省值为 1，当运行多个应用程序，并且在非常短的时间内，应用的事务请求大量提交时，增大 MINCOMMIT 的值，可以提高性能。另外，在增大 MINCOMMIT 时，还需要增大 LOGBUFSZ 参数，以避免日志缓冲区因写满而被强制写入磁盘。LOGBUFSZ 应该设为：

```
MINCOMMIT * (事务使用的日志空间的平均值)
```

13. 语句集中器参数

DB2 V9.7 推出了语句集中器功能，能自动分析动态 SQL 语句，对于只是常量不同的动态 SQL 语句，能让它们自动地共享含有参数标记的 SQL 语句，从而避免修改应用程序。于是，这些相似的 SQL 语句都能命中高速缓存，共享访问计划，节省了编译时间和 CPU 开销。

缺省情况下，语句集中器处于禁用状态。如果希望对所有动态语句启用语句集中器，只需将 STMT_CONC 这个数据库配置参数设置为 LITERALS，如下所示：

```
db2 update db cfg using STMT_CONC LITERALS
```

7.1.4　注册变量优化

1. DB2_PARALLEL_IO 注册变量

对于含有多个磁盘的容器很有用。默认情况下，DB2 对一个表空间容器一次只发出一个预取请求。但是如果容器含有多个磁盘，实际上应该对它同时发送多个预取请求。最简单用法如下，意思是所有表空间都将使用 6 作为每个容器的磁盘数，其中 6 为默认磁盘数。

```
DB2_PARALLEL_IO=*
```

如果容器在一个 5 块硬盘的 RAID-5（4D+1P）上，则可以设置 B2_PARALLEL_IO 为“*:4”，其中虽然是 5 块盘，但是 1 块用于校验，只有 4 块用来存放数据。

```
DB2_PARALLEL_IO=*:4
```

2. DB2CODEPAGE 注册变量

通常 DB2 会自动从操作系统读取该代码页信息，所以不需要设置 DB2CODEPAGE。但是

如果服务器和客户端代码页不匹配时，可以通过 db2set 命令来修改该值，执行下面的命令可以将该 DB2CODEPAGE 设置为 1208（表示 Unicode 代码页）：

```
db2set db2codepage=1208
```

3. DB2COMM 注册变量

通过 DB2COMM 注册表变量为当前 DB2 实例设置通信协议。如果该注册表变量未定义或设置为空，那么启动数据库管理器时不会启动任何协议连接管理器。通过如下命令设置通过 TCP/IP 连接到当前 DB2：

```
db2set DB2COMM=TCPIP
```

小技巧：查询和更改访问 DB2 的端口号

当 DB2COMM 设置为 TCPIP 时，可以通过 TC/PIP 访问数据库，其默认的端口号为 50000，要修改或者查看这个端口号，可以查看 DBM 参数 SVCENAME 值，例如 db2c_inst1，然后在 services 文件中找到键 db2c_inst1，其值（50000/tcp）就是访问的端口号，通过将 50000 修改成其他的值，例如 50005，然后重启实例，就可以将访问端口更改为 50005。对于 Windows 操作系统，services 文件位于 C:\Windows\System32\drivers\etc 目录下，而对于 Unix 或者 Linux 操作系统，services 文件位于/etc 目录下。

7.1.5 常见问题总结

参数设置不当会引起很多性能问题，甚至是严重的系统瓶颈。下面是需要特别注意的地方：

（1）磁盘的吞吐量取决于磁盘个数。那些具有大型用户表或临时表的表空间，应该尽量分散到所有磁盘。

（2）缓冲池可以占用可用内存的大约 75%（OLTP）或 50%（OLAP）。

（3）应该增加 SORTHEAP，以避免排序溢出。

（4）在高性能的独立磁盘上存放本地事务日志。

（5）在数据仓库应用中，如果数据量超过 300GB，应考虑使用 DPF 分区功能。使用数据库配置建议程序，可以获得良好的初始配置设置。

（6）对 DPF 环境，使用本地文件，而不是 NFS 文件作为诊断日志路径。

7.2 日常运维工具的使用与优化

数据库运维就像一场持久战，必须熟读兵法，并且根据实际战局的发展，进行灵活的运用，才能取得最后战役的胜利。

7.2.1 知已知彼——统计信息收集

近年来荧屏上《借枪》、《潜伏》、《黎明之前》等谍战片热播，引起人们的极大兴趣，娱乐之余，也说明了一个事实，情报工作对于战争的胜负起着至关重要的作用。对于 DB2 而言，

收集情报、统计信息也同样重要，优化器需要统计信息生成访问计划，不准确的统计信息就像战争中的假情报一样，会让优化器选择错误的访问计划，从而影响查询的性能。

统计信息收集的命令比较简单，例如可以使用下面的命令对表 EMPLOYEE 执行 RUNSTATS 操作。这个命令将更新 EMPLOYEE 表及其上的所有索引的统计信息。

```
db2 "runstats on table EMPLOYEE with distribution and detailed index all"
```

通常在以下情况，需要进行统计信息的收集。

- 当表的数据经过大量的增删改之后。
- 当表经过重组之后。
- 当创建新的索引之后。
- 运行 bind 命令之前。
- 当 prefectch 参数改变之后。

由于统计信息的收集同样需要消耗资源，因此需要快速而高效地收集统计信息，从而尽可能降低统计信息收集对于系统的影响。

为了避免不必要的统计信息收集，只对没有收集或者较长时间没有收集统计信息的表或者索引收集统计信息，使用下面的语句可以找出没有收集过统计信息的表：

```
db2 "select tabname from syscat.tables where stats_time is null"
```

使用下面的语句查找没有收集过统计信息的索引：

```
db2 "select indname from syscat.indexes where stats_time is null"
```

使用下面的语句分别可以查到过去 15 天内没有更新过统计信息的表和索引：

```
db2 "select tabname from syscat.tables where stats_time < current timestamp - 15 days"
db2 "select indname from syscat.indexes where stats_time < current timestamp - 15 days"
```

对于以上查询结果中的表或者索引执行 RUNSTATS 来更新统计信息，以使 DB2 的查询优化器能够选择最优的查询访问计划。

统计数据收集的速度与目标表的数据量、目标表上的列数目、目标表上的索引数目以及统计信息的详细程度有着极大关系。因此为了优化 runstats，需遵循以下原则：

（1）只对需要的列收集统计信息，对于 SQL 语句中出现在 WHERE 子句中的列，或者 JOIN 条件中的列以及 GROUP BY/ORDER BY 等子句中的列收集统计信息。

（2）创建索引的同时收集统计信息，并且根据数据量和查询的需要选择 DETAILED 或者 SAMPLED 参数。

（3）使用 LOAD 命令的 REPLACE 选项向表中加载数据时，指定 STATISTICS USE PROFILE 参数，在加载数据的同时，收集统计信息。

（4）对于较大的非关键的表使用 SAMPLED 参数，能够加快 RUNSTATS 执行速度。

7.2.2　集中兵力——碎片整理工具

数据库运行一段时间后，当其中的表经过大量插入（Insert）、更新（Update）、删除（Delete）等操作后，数据分布就会非常分散，从而会增加 I/O 操作，降低系统的性能，因此需要对表及

其索引进行重组（即执行 REORG 操作）将数据重新组织。因为 REORG 操作非常费时，所以 DB2 提供了 REORGECHK 命令，用于检查是否应该进行 REORG 操作。在整个运维期间，要定期地对表执行 REORGCHK 命令。REORGCHK 首先调用 RUNSTATS，更新表的统计信息，然后利用这些新的统计信息，通过一些预先定义的公式（在命令的输出里面可以看到这些公式）进行计算，来判断表是否需要重组。

下面的命令输出中，分别显示了表和索引的统计信息，以及通过公式计算出的值。如果计算出的值达到阈值的要求，例如，表 TEST 的空间利用率的百分比 F2 为 40，小于阈值 70，对应的位置就会显示“*”，否则就会显示“－”。如果对应的 REORG 列出现一个或多个“*”时，就说明应该考虑对该表或索引进行重组操作。

```
=> db2 reorgchk update statistics on table all

Doing RUNSTATS ....


Table statistics:

F1: 100 * OVERFLOW / CARD < 5                               ←计算公式和阈值
F2: 100 * (Effective Space Utilization of Data Pages) > 70
F3: 100 * (Required Pages / Total Pages) > 80

SCHEMA.NAME          CARD    OV    NP    FP ACTBLK    TSIZE  F1  F2  F3 REORG
---------------------------------------------------------------------------
Table: IIFVT14.TEST
                        4     0     1     1      -       60   0  40 100  -*-
Table: IIFVT14.NEWTEST
                   196620     0   982  1024      -  3932400   0  96  96  ---

......
------------ -------------------------------------------------------------

Index statistics:

F4: CLUSTERRATIO or normalized CLUSTERFACTOR > 80

F5: 100 * (Space used on leaf pages /
    Space available on non-empty leaf pages) > MIN(50, (100 - PCTFREE))

F6: (100 - PCTFREE) * (Amount of space available in an index with one less level
   / Amount of space required for all keys) < 100

F7: 100 * (Number of pseudo-deleted RIDs / Total number of RIDs) < 20

F8: 100 * (Number of pseudo-empty leaf pages / Total number of leaf pages) < 20

SCHEMA.NAME               INDCARD    F4  F5  F6  F7  F8 REORG
-------------------------------------------------------------
Table: IIFVT14.NEWTEST
Index: IIFVT14.NI
                           196620    98  92   -   0   0 -----
......
---------------------------------------------------------------------------
```

对于需要重组的表（即需要执行 REORG 的表），通常选择在系统的维护窗口来执行重组，并需要在维护窗口结束前做完。

使用下面的 REORG 命令对表 TEST 及其索引进行实际的重组：

```
#对表 NEWTTEST 进行重组（表上的索引将自动被重组）
db2 reorg table  TEST
#只重组表 NEWTEST 的索引
db2 reorg indexes all for table TEST
```

以下对于 REORG 工具的优化和建议可以加快 REORG 的进程，并减小 REORG 期间所造成的影响：

（1）通常 REORG 耗时很长，而且 REORG 期间用户一般不能访问表，所以在 REORG 前，应该通过分析 REORGCHK 命令的结果，只对需要 REORG 的表或索引进行 REORG。

（2）对于从来不使用的索引，应将其删除，以节省 REORG 的成本。

（3）REORG 期间会使用临时表空间，因此将临时表空间定义在性能较好的存储上能够提高 REORG 的速度。

（4）对于关键业务系统，可以使用 ONLINE REORG，并实施监控，以便在出现性能问题时能够终止 REORG 操作。

（5）对表进行 OFFLINE REORG 后，无须再对索引进行 REORG，因为 OFFLINE REORG 后，会重建相应的索引。

（6）REORG 后，需要通过 RUNSTATS 来更新统计，然后通过 REBIND 重新绑定静态 SQL 语句，以使 DB2 优化器能够得到最新最准确的信息。

（7）离线 REORG 中途失败，有可能导致索引数据的损坏，因此尽量不要强行终止正在运行的离线 REORG 操作。

注意：自动维护

除了手工维护外，DB2 也支持自动维护，包括自动统计信息收集（AUTO_RUNSTATS）、自动碎片整理（AUTO_REORG）和自动备份（AUTO_BACKUP）。

DBA 可以根据自己的实际情况来定义维护窗口政策文件（格式为 XML），随后调用存储过程 AUTOMAINT_SET_POLICYFILE 和 AUTOMAINT_GET_POLICYFILE 分别进行政策文件的设置和获取。

7.2.3　随机应变——重新绑定

当重新收集统计信息之后，DB2 优化器会根据最新的信息调整执行计划，后续查询编译将使用新统计信息。但是对于静态语句，就像得到作战指令已经派出的部队，如果不重新发布指令，是不会起作用的。类似的，当数据库的执行环境发生变化时，例如对该静态 SQL 访问的表上增加了索引，由于静态 SQL 不会重新生成访问计划，就不会使用该索引，从而达不到最佳的执行效率，这个时候就需要进行重新绑定（REBIND）来适应环境的变化。

比较：动态 SQL 和静态 SQL 的区别

在 DB2 中，动态 SQL 语句是在应用程序运行时被编译和执行的，而静态 SQL 语句的编译是在应用程序运行前进行的，编译的结果会存储在 DB2 的包（PACKAGE）中。当程序运行时，数据库将直接执行编译好的 SQL 语句，降低运行时的开销，从而提高执行效率。

通过 BIND 命令，DB2 可以将静态 SQL 语句的执行计划重新绑定，这样在执行的时候，就会按照保存在包中的访问计划执行，避免了重复的编译和访问计划的生成。

对于 SQC（嵌入 C）等程序，在 BIND 的时候会创建一个包含静态 SQL 的包。而存储过程等例程在部署时，也会生成相应的包。通过 IBM 提供的 pureQuery 技术，还可以将传统的动态 SQL 绑定到相应的包上，这样就可以转变为静态 SQL 来执行。

使用 REBIND 命令时，需要提供包名。通过 LIST PACKAGES 命令可以查看包名（**Package 列**）及相关信息：

```
#List packages 的相应语法
DB2 LIST PACKAGES [FOR {USER | ALL | SYSTEM | SCHEMA schema-name}] [SHOW DETAIL]

#查看当前用户所有包的详细信息的
=> db2 list packages show detail

 Package        Schema    Bound by   Valid    Isolation level   Blocking
-----------    -------   --------   -------  --------------    ---
P1047178439    DB2INST1  DB2INST1   Y         CS                U
P3347436437    DB2INST1  DB2INST1   Y         CS                U
```

对于指定的单个包进行重新绑定时，使用 REBIND 命令：

```
=>  db2 rebind package DB2INST1.P1047178439
DB20000I  The REBIND PACKAGE command completed successfully.
```

但是如果对所有的包重新绑定，就需要使用 db2rbind 命令了。

```
[db2inst1@db2 ~]$ db2rbind sample -l db2rbind.log all
 Rebind done successfully for database 'SAMPLE'.
```

动态 SQL 在执行时编译，其编译结果也会存储到包缓存（PACKAGE CACHE）中。当更新了统计信息时，要使缓存中的动态语句使用最新的统计信息，可以通过 FLUSH PACKAGE CACHE DYNAMIC 更新包缓存。

7.3 数据移动工具

在战争中，快速转移部队的能力通常是致胜的法宝之一，对于数据库来说，快速地移动数据同样至关重要。针对不同的应用场景，DB2 提供了如表 7-2 所示的工具：

表 7-2 数据移动的工具

命令	功能	特点
EXPORT	将数据从数据库中导出到文件中	使用简单，但速度较慢
IMPORT	将 EXPORT 导出的文件导入到数据库中	速度较慢，使用比较简单，会激发相应的触发器动作
LOAD	将 EXPORT 导出的文件或者游标中的数据导入到数据库中	速度很快，但是使用较复杂，不会激发触发器动作
INGEST	将 EXPORT 导出的文件导入到数据库中	可以快速移动大量实时数据而不必锁定目标表，在性能和可用性之间做到了极好的平衡
db2move	和 EXPORT，IMPORT，LOA，COPY 等选项配合使用能够在数据库间批量移动或复制大量的数据库对象	批量复制对象和数据
db2relocatedb	更改数据库名称或重定位数据库或者数据库的组件	通过更改数据库元数据实现，只能离线操作
ADMIN_MOVE_TABLE	在线或离线移动表	只能在同一数据库内移动
ADMIN_COPY_SCHEMA	批量复制对象	只能在同一数据库中复制

7.3.1 导出数据（EXPORT）

EXPORT 命令简单来讲就是通过查询语句选择要导出的数据并保存到 DEL、IXF 和 WSF 等格式的文件中。接下来，看看如何使用 EXPORT 命令导出数据。

如下所示，OF DEL 指明了要将数据输出到 DEL 格式的文件 EMPLOYEE_RESUME_PHOTO.DEL 中，而 LOBS 类型数据导入到路径/data1/lobs 和/data2/lobs 目录下，XML 数据导入到/data3/xml 和/data4/xml 目录下，并将导出过程中的消息存入消息文件 msg.out 中。

```
    [db2inst1@db2 ~]$db2 "EXPORT TO EMPLOYEE_RESUME_PHOTO.DEL OF DEL LOBS TO
/data1/lobs,/data2/lobs XML TO /data3/xml,/data4/xml MESSAGES msg.out SELECT
R.EMPNO,R.RESUME_FORMAT,R.RESUME,P.PHOTO_FORMAT,P.PICTURE FROM EMP_ RESUME R,EMP_PHOTO
P WHERE R.EMPNO=P.EMPNO"

    Number of rows exported: 16
```

从上面的例子可以看出 EXPORT 的使用非常简单，但是如果就此以为完全掌握了 EXPORT 的使用，在实际的应用中必然会遇到种种问题，下面就是一些使用 EXPORT 命令的注意事项与技巧。

（1）使用 EXPORT 命令，必须具有相应权限。由于导出操作需要使用 SELECT 语句，因此使用 EXPORT 命令需要有 SYSADM 或者 DBADM 管理权限，或者对导出的表有 CONTROL 或 SELECT 特权。

（2）使用 EXPORT 命令可以导出多种文件格式，但每种格式都有各自的特点，如果需要在导出数据的同时也导出表的定义，需要选择 IXF 格式，使用这种格式导出的文件，在导入的时候，如果目标表不存在，可以自动创建相应的目标表。通常这种格式导出的文件在导入时，效率比其他格式要高。而其他文件格式则只能导出相应的数据，导入时，目标表必须实际存在。

（3）要提高导出数据的效率，必须从两方面着手，首先，过滤数据的查询语句的效率直接决定了导出数据效率，因此必须优化查询语句，其次对于大数据和 XML 数据，可以指定多个文件路径并行导出，从而提高效率。

（4）另外 EXPORT 是基于查询的导出，其导出数据必须经过数据库引擎，当导出海量数据时，其导出数据的效率是有局限性的，那有没有什么办法直接从数据文件抽取数据呢，答案是肯定的，High Performance Unload 就是这样一款工具，它不需要经过数据库引擎，直接从数据库文件中抽取数据，从而能够以数倍于 EXPORT 的速度导出海量数据。

7.3.2 导入数据（IMPORT）

DB2 中导入数据的方法在 DB2 V10 以前主要有 2 种，一种是 IMPORT，另一种是 LOAD。IMPORT 相对于 LOAD 来说比较简单，可以理解为 EXPORT 的逆过程。在这里，先重点介绍 IMPORT 命令的使用方法。

IMPORT 命令使用 INSERT 语句来批量插入数据，因此，IMPORT 会遵循 INSERT 语句的各种限制，例如在插入时需要验证各种约束、触发器和参照完整性检查等。

我们首先看一下 IMPORT 命令格式：

```
IMPORT FROM filename OF {IXF | ASC | DEL | WSF}
[LOBS FROM lob-path [ {,lob-path} ... ] ]
[XML FROM xml-path [ {,xml-path} ... ] ][MODIFIED BY filetype-mod ...]
[METHOD {L ( col-start col-end [ {,col-start col-end} ... ] )
[NULL INDICATORS (col-position [ {,col-position} ... ] )] |
N ( col-name [ {,col-name} ... ] ) |
P ( col-position  [ {,col-position} ... ] )}]
[XMLPARSE {STRIP | PRESERVE} WHITESPACE]
[XMLVALIDATE USING {XDS [DEFAULT schema-sqlid]
[IGNORE (schema-sqlid [ {,schema-sqlid} ... ])]
[MAP ((schema-sqlid,schema-sqlid) [ {(schema-sqlid,schema-sqlid)} ... ])] |
SCHEMA schema-sqlid | SCHEMALOCATION HINTS }]
[ALLOW {NO | WRITE} ACCESS]
[COMMITCOUNT {n | AUTOMATIC}] [{RESTARTCOUNT | SKIPCOUNT} n]
[ROWCOUNT n] [WARNINGCOUNT n] [NOTIMEOUT] [MESSAGES message-file]
{{INSERT | INSERT_UPDATE | REPLACE | REPLACE_CREATE}
INTO {table-name [( insert-column , ... )] | hierarchy-description}
| CREATE INTO {table-name [( insert-column , ... )] |
hierarchy-description {AS ROOT TABLE | UNDER sub-table-name}
[IN tablespace-name [INDEX IN tablespace-name] [LONG IN tablespace-name]]}
```

IMPORT 所支持的导入文件类型包括 ASC、DEL、IXF 和 WSF。在 IMPORT 过程中产生的消息记录在 message_file 中。target_table_name 指定了目标表的名字。

1. 导入方式选项

我们可以通过不同选项来决定导入数据的方式：

（1）INSERT 选项：表示导入数据时不影响已经存在的数据，如果导入的行与已存在行有主键冲突，则本行不插入。

（2）INSERT_UPDATE 选项：此选项只针对有主键的表，在导入数据时需要比对主键，主键重复的话就 update（用新数据替掉原数据），否则就 insert（直接插入）。例如：表 test_tab，id 字段为主键，要更新的字段为 name。要导入的文件内容为：

```
101,'John'
102,'Tom'
```

导入语句是：

```
IMPORT FROM member.del OF DEL insert_update into test_tab
```

如果 test_tab 表中已经有一行数据的 id 字段值为 101，那么这行数据的 name 就会被更新为 John。如果没有 id 字段值为 101 的行，那么直接执行 insert 操作，插入一行数据（id=101，name=“John”）。

（3）REPLACE 选项：表示删除掉所有已经存在的行，并导入新的数据。既然会删掉原有的数据，那么这个操作是有风险的，使用时需要注意。

（4）REPLACE_CREATE 选项：如果目标表存在，则与 REPLACE 选项一样，如果目标表未定义，则建立目标表及索引，使用这个选项的前提是导入文件为 PC/IXF 格式。

（5）CREATE 选项：建立目标表及索引，并导入数据，使用这个选项的前提是导入文件为 PC/IXF 格式。

2. 导入时的权限要求

用户需要注意：使用 IMPORT 命令需要 SYSADM 或者 DBADM 管理权限，或者对要导入的表或视图拥有 SELECT、INSERT、CONTROL 或 CREATETAB 特权。

3. 并发控制选项

在 IMPORT 时，默认并发控制选项是 ALLOW NO ACCESS，也就是说，要对表加上互斥锁（X），因此，在导入过程中，表数据是不能读写的。如果希望在导入过程中对表可以读写，则需要使用 ALLOW WRITE ACCESS 选项，这时对表加 IX 锁。使用 ALLOW WRITE ACCESS 选项，即使没有设置 COMMITCOUNT，其行为也会和设置了 COMMITCOUNT AUTOMATIC 一样。需要注意的是，如果导入选项使用的是 REPLACE、CREATE 或者 REPLACE_CREATE，则不能使用 ALLOW WRITE ACCESS 选项。

4. 提交时机选项

使用 IMPORT 意味着执行批量记录的插入操作，默认情况下这些批量插入操作是在一个事务内进行的。由于在事务执行过程中会生成大量的在线日志，因此需要保证足够的在线日志空间；另外，在插入过程中会申请大量锁资源，可能导致锁升级。为了避免这种情况，可以使用 COMMITCOUNT 选项指定插入多少条记录就提交一次，或者使用 COMMITCOUNT

AUTOMATIC 选项来由 IMPORT 自己决定提交时机。另外，ROWCOUNT 选项被用来标明要导入多少行数据。

在下面的例子中，每插入 1000 条数据就提交一次。如果在 IMPORT 过程中失败，可以查看 msg.out 文件，找到最后一个插入成功并且已经提交的记录，随后再继续 IMPORT 的时候使用选项 RESTARTCOUNT，从失败的记录开始继续导入：

```
IMPORT FROM customer.ixf OF IXF
    COMMITCOUNT 1000
    MESSAGES msg.out
    INSERT INTO customer
```

如果使用 RESTARTCOUNT 20000 跳过前 20000 条数据继续导入，用 ROWCOUNT 选项指定要导入 30000 条数据，采用的命令如下所示：

```
IMPORT FROM myfile.ixf OF IXF
    COMMITCOUNT 1000 RESTARTCOUNT 20000 ROWCOUNT 30000
    MESSAGES msg.out
    INSERT INTO newtable
```

5. 导入数据时代码页的选择

最后，来看一个比较综合的例子，比如已经有一个从数据库里导出的文件 employee.del，它使用的代码页为 1386，一共 42 行数据。执行下面的导入操作：

```
[db2inst1@db2 ~]$db2 "IMPORT FROM employee.del OF DEL MODIFIED BY codepage=1208
dateforma
t=\"YYYYMMDD\" METHOD P (1, 2, 11, 12) ALLOW NO ACCESS COMMITCOUNT AUTOMATIC RES
TARTCOUNT 41 ROWCOUNT 1 MESSAGES msg.out INSERT INTO myemployee (EMPNO, FIRSTNME
, BIRTHDATE, SALARY)"

Number of rows read         = 42
Number of rows skipped      = 41
Number of rows inserted     = 1
Number of rows updated      = 0
Number of rows rejected     = 0
Number of rows committed    = 42
```

上面的导入操作中，通过 METHOD P 指定只导入文件中的第 1、2、11 和 12 字段，ALLOW NO ACCESS 表示导入过程中不能读写，COMMITCOUNT AUTOMATIC 表示提交频率由系统决定，此次将跳过前 41 行，而从 42 行开始导入，ROWCOUNT 表示只导入一行。接着再看 MODIFIED BY 里面的选项，codepage=1208 表示导入文件的代码页是 1208，dataformat= \“YYYYMMDD\”表明了导入文件中日期字段的格式。目前看来一切正常，最后来看一下导入后的表内容：

```
[db2inst1@db2 ~]$db2 select * from myemployee

EMPNO  FIRSTNME     BIRTHDATE  SALARY
------ ------------ ---------- -----------
200340              1956-05-17   31840.00

  1 record(s) selected.
```

从结果看确实成功地导入了一行数据。不对！FIRSTNME 字段怎么没有值？仔细看导入命令就会发现问题出在 codepage=1208 上，还记得导出文件使用的代码页是 1386 吗？其实，DB2 本身具备将导入文件代码页转化为自身代码页的能力，但前提是要告诉它导入文件的真实代码页，否则，导入后会出现字段值错误或者乱码现象。这次把导入命令中的 codepage 修改为 1386 之后再次执行：

```
    [db2inst1@db2 ~]$db2 "IMPORT FROM employee.del OF DEL MODIFIED BY codepage=1386 
dateforma
    t=\"YYYYMMDD\" METHOD P (1, 2, 11, 12) ALLOW NO ACCESS COMMITCOUNT AUTOMATIC RES
    TARTCOUNT 41 ROWCOUNT 1 MESSAGES msg.out INSERT INTO myemployee (EMPNO, FIRSTNME
    , BIRTHDATE, SALARY)"

    Number of rows read         = 42
    Number of rows skipped      = 41
    Number of rows inserted     = 1
    Number of rows updated      = 0
    Number of rows rejected     = 0
    Number of rows committed    = 42
```

然后再查询表的内容就可以看到正确的结果了。

```
    [db2inst1@db2 ~]$db2 select * from myemployee

    EMPNO  FIRSTNME     BIRTHDATE  SALARY
    ------ ------------ ---------- -----------
    200340 刘          1956-05-17   31840.00

      1 record(s) selected.
```

通过上面的例子，笔者的建议是在导出导入时显式指定 codepage，从而把命运掌握在自己手里。这一点在跨服务器导出导入的场景中尤为重要。当然，如果系统编码相对单一时，也可以通过 db2set db2codepage 将默认值更改为所需要的编码，从而避免每次都需要重复指定。

7.3.3　LOAD——导入大数据的核武器

如果说 IMPORT 是导入数据的常规武器，那么 LOAD 就是导入数据的核武器，对于 OLAP 系统，面对 TB 级的数据时，IMPORT 就很难满足性能的需求，这个时候就是 LOAD 大显身手的舞台了。LOAD 导入与 IMPORT 不同，它并不是通过 SQL 语句 INSERT 进行导入，而是把导入数据以数据页的形式直接存入数据库中。这就意味着摆脱了 SQL 语句的束缚，以一种物理方式把数据放入数据库中，并且在导入过程中不记录日志或者只记录少量的日志，特别适合大数据量导入。如图 7-1 所示，LOAD 在导入过程中只进行唯一性检查和非法值检查，不会进行完整性检查，不会调用触发器，这些逻辑层面的约束检查需要在导入之后手动执行 SET INTEGRITY 命令进行检查，因此对于存在约束定义的表，在导入后别忘了还需要约束检查这一步。

LOAD 导入的流程如图 7-1 所示：

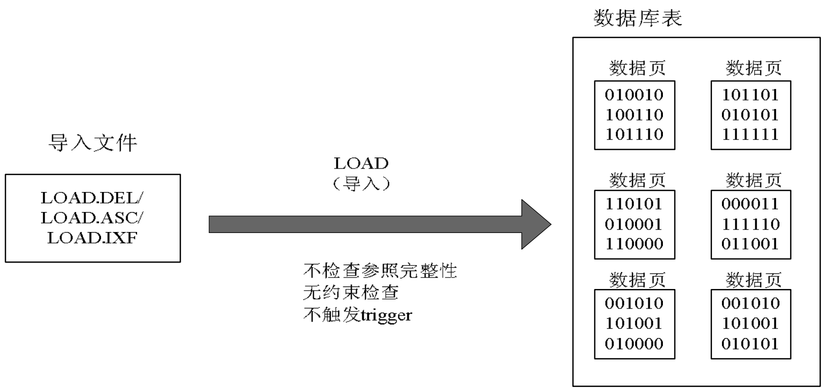


图 7-1　LOAD 导入示意图

LOAD 支持的数据格式包括 DEL、ASC、PC/IXF 和 CURSOR，其中 CURSOR 只能用在 LOAD 而不能用在 IMPORT，下一节将具体介绍 LOAD FROM CURSOR。

1. 使用 LOAD 命令导入数据

下面先通过一个例子来学习如何用 LOAD 进行导入。继续使用 IMPORT 中用到的导入文件 employee.del，只是这次改用 LOAD，操作结果如下：

```
=>db2 "LOAD FROM employee.del OF DEL MODIFIED BY codepage=1386 dateformat=
\"YYYYMMDD\" DUMPFILE=c:\load\emp.dmp METHOD P (1, 2, 11, 12) MESSAGES msg.out T
EMPFILES PATH c:\load INSERT INTO myemployee (EMPNO, FIRSTNME, BIRTHDATE, SALARY
) FOR EXCEPTION empexp NONRECOVERABLE INDEXING MODE REBUILD ALLOW NO ACCESS"

Number of rows read        = 42
Number of rows skipped      = 0
Number of rows loaded       = 42
Number of rows rejected     = 0
Number of rows deleted      = 0
Number of rows committed    = 42
```

上面例子中大部分选项和 IMPORT 导入是一样的。 需要注意的是，在 LOAD 导入中可以指定 DUMPFILE，DUMPFILE 用来存放在导入过程中的无效数据行。由于在导入过程中还会产生违例，违例表由选项 FOR EXCEPTION 决定，它是用户定义的表，用来存放违反唯一性约束的行。其表结构包括所有目标表的列、一个记录该行插入时间的时间戳列和一个存放违例原因的 CLOB 列。上例中违例表为 empexp，是在 LOAD 之前事先建立好的。建立过程如下：

```
=>db2 "CREATE TABLE empexp AS (SELECT myemployee.*, CURRENT TIMESTAMP AS T
IMESTAMP, cast ('' AS CLOB (32K)) AS MSG FROM myemployee) WITH NO DATA"
DB20000I  The SQL command completed successfully.
```

另外，在上面的 LOAD 实例中还使用了一些在 IMPORT 中没有见过的选项，接下来就深入研究一下 LOAD 的几个重要选项，从而使大家对 LOAD 有一个更全面的认识。

（1）LOAD 支持的加载方式

LOAD 的加载模式和上一节介绍的 IMPORT 类似。

- INSERT：直接插入数据而不影响原有的数据。

- REPLACE：删除掉所有现有数据，然后导入新的数据，这个选项还要求对相关的表有 DELETE 特权。
- TERMINATE：当 LOAD 操作出现异常时，可以用 LOAD…TERMINATE 操作来终止 LOAD 操作，并且系统会自动将被导入的表回滚至导入前的状态。
- RESTART：用来重启被终止的 LOAD 操作。将从最后一个一致点（导入成功的点）开始，继续未完成的操作。为了获取最后一个一致点，LOAD 会读取之前生成的各种文件，了解之前的导入进度和状态。用户可以通过 TEMPFILES PATH 指定临时文件存放位置，如果在 LOAD 命令中不进行指定，LOAD 会自动创建并保存在当前目录中。在导入成功后，这些临时文件会自动被删除。

（2）LOAD 支持的索引模式

很多人对 LOAD 导入过程中索引的使用有误区，其实这可以通过 INDEXING MODE 来指定导入过程中如何处置索引，支持的模式包括：

- REBUILD 模式：重建所有索引。
- INCREMENTAL 模式：在导入的同时对现有索引进行增量维护而不是重建。
- AUTOSELECT 模式：由 LOAD 依据索引的深度来自动选择是采用 REBUILD 还是 INCREMENTAL 方式。
- DEFERRED 模式：在 LOAD 过程中把涉及到的索引标记成需要重建，一般在数据库重新激活后或者第一次使用时再真正进行重建。

（3）LOAD 支持的并发机制

在 LOAD 过程中，会涉及到表的并发访问问题。LOAD 支持 ALLOW NO ACCESS 和 READ ACCESS ONLY 两种方式。

- ALLOW NO ACCESS 选项：默认情况下，LOAD 使用该选项，在导入过程中对表加互斥锁（X）。
- READ ACCESS ONLY 选项：使用该选项，在导入过程中对表加共享锁（S），对于已经存在的数据是可读的，但是新导入的数据必须在导入结束以后才可以访问。

另外，关于 LOCK WITH FORCE：如果某个应用和 LOAD 发生锁冲突而使 LOAD 处于等待状态的话，LOAD 必须要等待应用释放掉需要的锁，如果使用 LOCK WITH FORCE 选项，LOAD 会强制其他应用释放所需要的锁，这有点像 force applications。

（4）LOAD 对数据可恢复性的影响

在生产系统中为了保证数据库的可恢复性，将数据库置于归档模式是更加安全的方式。但是，用 LOAD 导入数据时不记录日志，虽然这样会对导入速度有好处，不过这也会产生问题，如果需要前滚恢复怎么办？数据库不可恢复怎么办？为了解决这个问题，在 LOAD 中可以指定一些选项来保证数据库的可恢复性。

LOAD 中有关可恢复性的选项有如下几种：

- COPY NO：这是默认选项，使用这个选项会导致表空间处于 BACKUP PENDING 状态（待备份状态），导入结束以后需要对表空间进行一次备份。相当于需要通过人为手段来保证数据库的可恢复性。
- COPY YES：建立一个导入数据的备份介质。在前滚恢复阶段，会用这个备份介质恢复

LOAD 导入的数据，相当于自动保证数据库的可恢复性。

- NONRECOVERABLE：表明此表是不能进行前滚恢复的，如果进行前滚恢复，此表会进入 Unavailable 状态。

2. 你所要知道的 LOAD 的四个阶段

如果要用好 LOAD，就需要了解 LOAD 的内部机制，首先应该了解 LOAD 过程中的不同阶段。LOAD 过程包含装载、构建、删除和索引复制 4 个阶段，这 4 个阶段是按顺序发生的，在一个阶段结束后才进入下一个阶段。在 LOAD 的过程中，如果目标表不包含唯一性索引，则不会执行完整的 4 个阶段，否则就进行完整的 4 个阶段。LOAD 的过程如图 7-2 所示：

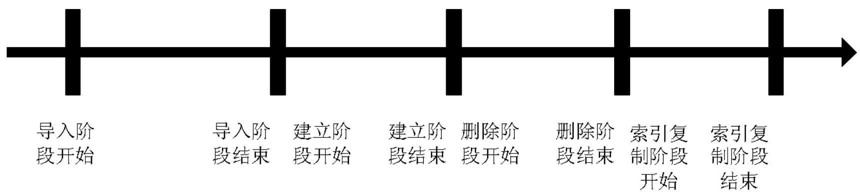


图 7-2　LOAD 阶段示意图

（1）导入阶段

在此阶段完成数据的导入，并收集表和索引的统计信息。SAVECOUNT 选项决定导入多少行会建立一个一致检查点，导入的消息被记录到 MESSAGE 文件中。如果导入中断后重新开始，LOAD 会从最近一次一致检查点开始导入。如果存在非法的行，比如违反非空约束的行或者格式完全不对的行，则该行会被记录到 DUMPFILE 中。

（2）建立阶段

如果导入表有索引的话，根据 INDEXING MODE 的设定建立相应索引。如果在 LOAD 中指定了 STATISTICS USE PROFILE，则会收集索引的统计信息。

（3）删除阶段

有人会问，不是导入数据了么，为什么还要再来个删除阶段呢？要知道，由于 LOAD 操作在导入效率上实属“批量”的 IMPORT 操作，速度很快，也就难免导入的数据中“鱼龙混杂”。这个删除阶段，就是要把违反唯一性约束或主键约束的行（比如导入员工信息，出现两个人的身份证号码是一样的）从表中删除，并放到违例表（Exception Table）中。如果未指定违例表，则违例行被直接删除，并且不留下任何追踪记录。

在 LOAD 失败时，我们可以通过 MESSAGE 文件调查导入时被拒绝的原因，并且对违例表中的行进行相应修改，然后用 INSERT 语句重新把违例表中的行插入到目标表中。

如果 LOAD 在删除阶段失败，可以使用 RESTART 选项重新 LOAD。另外，在删除阶段是记录日志的，因此如果有大量的违例行，可能会有日志空间满的风险。

（4）索引复制阶段

如果指定了 ALLOW READ ACCESS 并且在 LOAD 中使用了系统临时表空间，则在这个阶段会将索引从系统临时表空间复制到原表空间。在复制阶段，目标表会被设定为离线状态，不能访问。

3. LOAD 过程中表的状态以及应对方法

在 LOAD 过程中，涉及到的表会处于各种状态，通过这些状态对数据的访问进行限制和完整性保护。这样，当 LOAD 出现问题时，就能知道发生了什么，并且知道如何把表恢复到可用状态。

那么如何查询表的状态呢？其实很容易，只需要执行 LOAD QUERY 就可以完成，其中 table_name 是目标表名：

```
LOAD QUERY TABLE table_name
```

下面就来介绍一下 LOAD 的各种状态：

（1）Normal：正常状态，一般新建的表都是这种状态。

（2）Load in Progress：当表正在进行 LOAD 时，会处于这个状态。

（3）Load Pending：当表在 LOAD 过程中发生意外中断，表会处于这个状态，意即表被挂起了，需要使用 LOAD TERMINATE、LOAD RESTART 或者 LOAD REPLACE 命令来解除这种状态。

（4）Read Access Only：如果使用 LOAD 时指定了 ALLOW READ ACCESS（只读），表的状态信息会包括 Read Access Only。

（5）Unavailable：如果在 LOAD 过程中使用了 NONRECOVERABLE 选项，当使用 LOAD 之前的某个时间点的数据库备份进行数据库恢复时，如果前滚到这个 LOAD 操作结束后的某个时间点，表便被置于 Unavailable 状态，是不可以访问的。在这种情况下，要访问这个表的数据，只能重新使用数据库备份进行恢复且只能前滚到 LOAD 操作之前的某个时间点。

（6）Not Load Restartable：表处于一种不完整导入状态，不允许使用 LOAD RESTART 继续执行，只能通过 LOAD TERMINATE 或者 LOAD REPLACE 来解除这种状态。一般有两种情况会导致表处于这种状态，一种是 LOAD 失败，之后又前滚恢复到 LOAD 之后的某个时刻；另一种是在表处于 Load in Progress 或者 Load Pending 时进行了在线备份。这两种情况都会导致 LOAD RESTART 需要的相关状态信息不可靠，因此，DB2 使用 Not Load Restartable 状态来阻止 LOAD RESTART。

（7）Unknown：表处于未知状态，LOAD QUERY 无法获得表状态信息。当客户端版本与数据库服务器版本不同时才会出现这种状态。比如，如果使用一个 DB2 V9.7 的客户端去连接 DB2 V9.8 的数据库，由于 V9.8 的表状态和 V9.7 有所不同，如果表正好处于这种不同的状态，则使用 LOAD QUERY 会查询到 Unknown 状态。

（8）Set Integrity Pending：如果表上存在约束，如参照完整性约束，在使用 LOAD 时表会处于这个状态，在 LOAD 结束后必须通过执行 SET INTEGRITY（执行完整性检查）使表摆脱这种状态。

需要指出的是，表可能同时处于上述的多种状态，比如，如果导入时使用了 ALLOW READ ACCESS 选项，通过 LOAD QUERY 查询时会发现表同时处于 Load in Progress 和 Read Access Only 状态，但 Read Access Only 只是一个从属状态，如果导入结束，Read Access Only 会自动消失。

LOAD 之后如果需要解除一个或多个表的 Set Integrity Pending 状态，需要进行相应的手动

操作。基本方法如下，其中 table-name 为目标表名：

```
SET INTEGRITY FOR table-name IMMEDIATE CHECKED INCREMENTAL
```

通过上面的命令可以由系统立刻进行某些类型的一致性检查，检查后 CONST_CHECKED 字段的各字节会从 N 变成 Y。如果检查过程没有遇到任何违例，则表会摆脱 Set Integrity Pending 状态。如果检查过程中发现违例，则立刻报错中止（SQLSTATE 23514），表会继续保持在 Set Integrity Pending 状态。

当然，也可以使用 FOR EXCEPTION 选项来指定把违例行移动到违例表中。这里说的违例表（Exception Table）和在前面有关 LOAD 选项中提到的违例表是一个概念，因此，可以复用在 LOAD 中用过的违例表。例如可以指定把 SALES 表中的违例行放到违例表 sales_exceptions 中，这样 SALES 表将摆脱 Set Integrity Pending 状态：

```
SET INTEGRITY FOR SALES IMMEDIATE CHECKED
    FOR EXCEPTION IN SALES USE sales_exceptions
```

如果想要跳过某些类型的自动约束检查，可以使用 IMMEDIATE UNCHECKED 选项。可以通过下面命令跳过对表 SALES 的外键检查。执行完这个命令之后，CONST_CHECKED 字段的字节 1 会从 N 变成 U。

```
SET INTEGRITY FOR SALES FOREIGN KEY IMMEDIATE UNCHECKED
```

4. 多分区环境下的 LOAD

与单分区环境相比，在多分区环境下，可以用几乎完全相同的命令来实现导入，但是，两者之间还是存在比较大的差异。

在多分区环境下导入数据一般有两个阶段，第一个阶段是建立阶段，这个阶段会获得导入需要的资源，比如表的锁；第二个阶段是导入阶段，数据将被导入到相应分区的表中。

下面就用一个例子来说明：

```
LOAD FROM LOAD.DEL of DEL REPLACE INTO SALES
```

上面这条语句看起来很简单，而且在单分区和多分区环境里的含义是一样的，都是把 LOAD.DEL 里面的数据导入到 SALES 里。不过，它隐藏了在多分区环境下导入数据的内部过程，我们就从这条语句入手，来了解一下在多分区环境下导入数据的特点。

在多分区环境下，SALES 中的数据可能分散在不同的数据库分区，分区键决定了表的每行数据应该放在哪个数据库分区。这个工作最开始由 pre-partitioning agent 进程负责，它把数据分配（这里基本上是均匀分配）给不同分区上的 partition agent，partition agent 负责把数据分发给各个分区的 load agent，load agent 负责把数据导入到相应的分区。这里面涉及到不同分区间的通信，因此，在导入前要确保数据库管理器参数 svcename 和 DB2COMM 数据库环境变量设置正确。

各个 agent 之间的调用关系如图 7-3 所示：

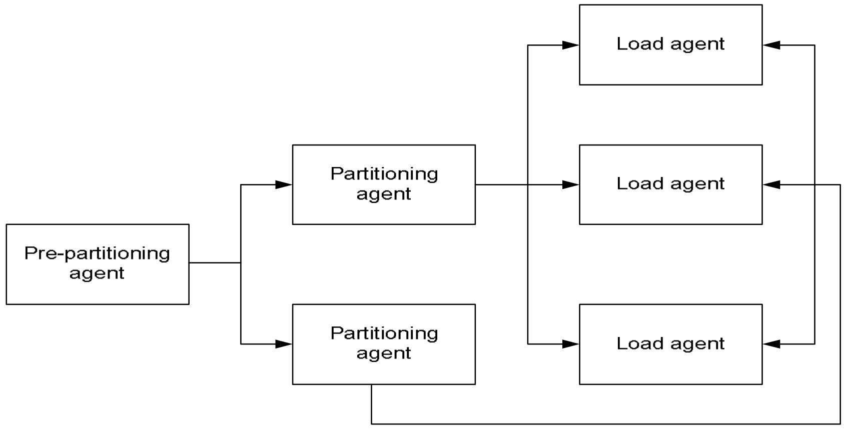


图 7-3　多分区环境下导入数据时所涉及的代理关系示意图

在上面向 SALES 表中导入数据的命令中省略了许多可选参数。实际上，在多分区环境下执行 LOAD 的时候，可以使用 LOAD 提供的参数 PARTITIONED DB CONFIG 来灵活控制，如下所示：

```
LOAD [CLIENT] FROM file OF {ASC | DEL | IXF | CURSOR}
...
{INSERT | REPLACE [{KEEPDICTIONARY | RESETDICTIONARY}] | RESTART | TERMINATE}
INTO table-name [( insert-column [ {,insert-column} ... ] )]
..
PARTITIONED DB CONFIG partitioned-db-option
PART_FILE_LOCATION x
OUTPUT_DBPARTNUMS x
PARTITIONING_DBPARTNUMS x
MODE x
MAX_NUM_PART_AGENTS x
ISOLATE_PART_ERRS x
STATUS_INTERVAL x
PORT_RANGE x
CHECK_TRUNCATION
MAP_FILE_INPUT x
MAP_FILE_OUTPUT x
TRACE x
NEWLINE
DISTFILE x
OMIT_HEADER
RUN_STAT_DBPARTNUM x
```

下面讲解 PARTITIONED DB CONFIG 提供的几个重要参数：

（1）分区参数

- OUTPUT_DBPARTNUMS：用来决定哪些分区参与到 LOAD 过程中。在上面命令中没有显式指定，即隐含规定了包含 SALES 的所有分区都参与 LOAD，需要注意的是 OUTPUT_ DBPARTNUMS 只能设置为包含导入表的分区的子集。
- PARTITIONING_DBPARTNUMS：指定了哪些分区负责分布数据。上面命令中没有显式指定，则由 LOAD 确定由多少分区以及在哪个分区上启用 partition agent 来负责分布数据。

（2）mode 参数

它决定了导入的行为，主要包括：

- PARTITION_AND_LOAD：数据被同时分布到各个分区并执行导入。
- PARTITION_ONLY：数据被分布到各个分区，生成导入文件，但不执行导入。导入文件包括一个文件头，描述了数据是如何在各个分区分布的。当需要导入时，可以使用 LOAD_ONLY 选项把这些已经分布好的数据导入到各个分区。
- LOAD_ONLY：如果已经通过 PARTITION_ONLY 把数据分布到了各个分区，这时可以用 LOAD_ONLY 选项把数据导入到各个分区中。
- LOAD_ONLY_VERIFY_PART：如果数据已经被分布到了不同的数据库分区，数据在导入过程中，会对每行数据进行验证，如果出现了违反分区键的违例数据，这些数据被放到 dump 文件中，如果没有指定 dump 文件，这些数据被丢弃，这些信息会被记录到相应分区的 MESSAGE 文件里。
- ANALYZE：用来生成最佳分布图（在所有数据库分区之间均匀地分布数据）。

如果像上面的命令那样执行导入，会默认使用 PARTITION_AND_LOAD 模式，

（3）ISOLATE_PART_ERRS 参数

在执行 LOAD 过程中可以指定 ISOLATE_PART_ERRS 选项来控制如何处理导入过程中的错误。ISOLATE_PART_ERRS 一般情况下默认值为 LOAD_ERRS_ONLY，如果使用了 LOAD 命令的 ALLOW READ ACCESS 和 COPY YES 选项，ISOLATE_PART_ERRS 的默认值则为 NO_ISOLATION。

ISOLATE_PART_ERRS 的值和含义如下：

- SETUP_ERRS_ONLY：如果在建立阶段出现错误，例如分区、表空间或表不可访问，只有出错分区停止执行，其他分区照常导入。如果在导入阶段出现错误，则整个 LOAD 操作都将失败。
- LOAD_ERRS_ONLY：如果在导入过程中某个分区失败，其他分区的导入会继续，但是导入的数据只有在执行 LOAD RESTART 成功之后才可见。在建立阶段出现的任何错误都会导致整个 LOAD 失败。
- SETUP_AND_LOAD_ERRS：无论在建立或者导入阶段出现错误，都只影响出错的分区，其他分区照常执行，但是导入的数据在全部成功导入之前是不可见的。这个选项只能用在 ALLOW READ ACCESS 和 COPY YES 被指定的情况下。
- NO_ISOLATION：无论任何阶段的错误都会导致 LOAD 全部中止。

介绍了这么多有关 PARTITIONED DB CONFIG 包含的选项，再来看一个在 LOAD 命令中显式使用 PARTITIONED DB CONFIG 选项的例子：

```
LOAD FROM LOAD.DEL of DEL REPLACE INTO SALES
            PARTITIONED DB CONFIG PARTITIONING_DBPARTNUMS (3,4)
```

在这个例子中，从分区 2 执行了上述命令，使用了 PARTITIONING_DBPARTNUMS 选项，它指定分区 3 和 4 各产生一个 partition agent 来负责分布数据到相应分区并进行导入。相关进程如图 7-4 所示：

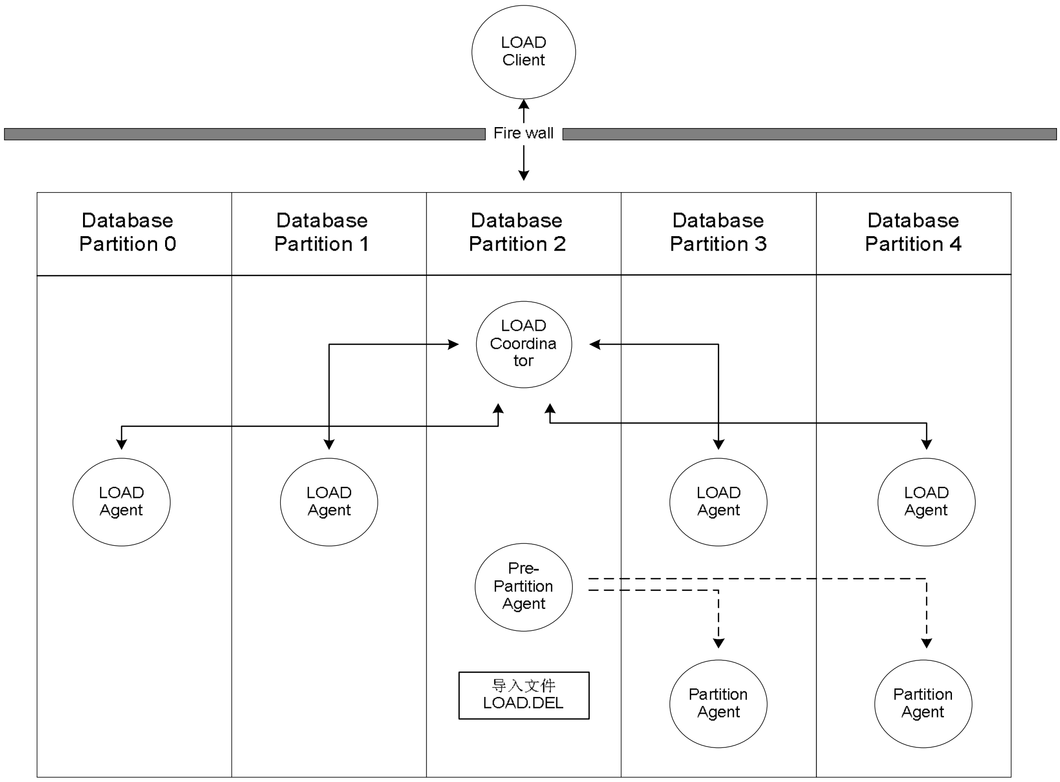


图 7-4 数据分发给 3 号和 4 号分区的代理来进行导入

接下再来看一个只分布数据而不导入的例子，仍然从 2 号分区执行下面的命令：

```
LOAD FROM LOAD.DEL OF DEL REPLACE INTO SALES
                PARTITIONED DB CONFIG MODE PARTITION_ONLY
                                     PART_FILE_LOCATION /db2/data
                                     OUTPUT_DBPARTNUMS (1,3)
```

在这个例子里指定了 MODE PARTITION_ONLY 选项，代表只分布数据不进行导入，OUTPUT_DBPARTNUMS（1，3）代表由分区 1 和分区 3 来负责导入（这里只是生成需要导入的数据），导入数据文件放置在由 PART_FILE_LOCATION 指定的/db2/data 路径下，文件名字由原来导入文件名字 LOAD.DEL 加上分区号，比如/db2/data/LOAD.DEL.001。图 7-5 展示了上述过程。

比较：单分区和多分区下 LOAD 的区别

通过上面的几个例子，可以了解在多分区环境下和单分区环境下数据导入的主要差别在于多分区环境下 LOAD 需要有一个数据分布的过程。当然，也有一些相同点，例如完成分布后，每个分区上的导入操作和单分区数据库的导入是一样的，也是通过直接装入数据页的办法实现的，例如在单分区上的一些命令选项在多分区环境下也适用，比如出错以后同样可以用 RESTART 选项重新执行 LOAD，也可以用 LOAD QUERY 查询当前表的 LOAD 状态。

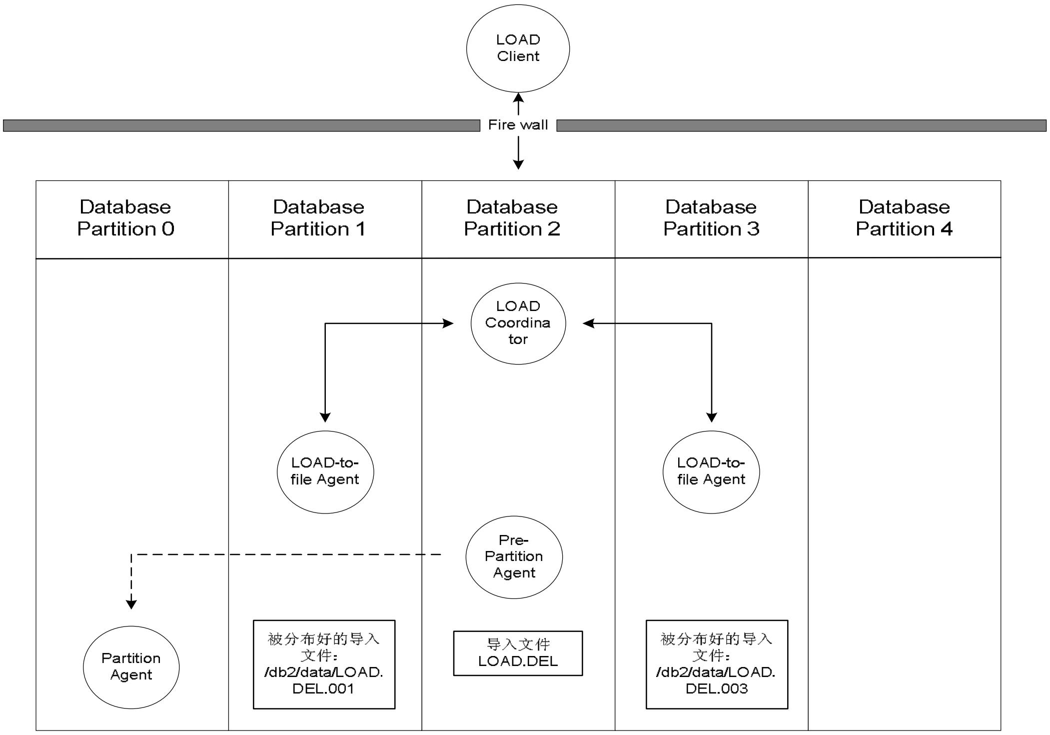


图 7-5　只分布数据到 1 号和 3 号分区

5. LOAD 最佳实践

LOAD 一般用在需要导入大量数据的场合，由于其速度远比 IMPORT 快，但是不当的 LOAD 提速操作，会引起性能问题，可能会使初衷折回原点，甚至引起事故。

决定 LOAD 操作的性能因素有多个，包括数据量、索引数量以及导入选项等。要提高 LOAD 的速度，一般在 LOAD 方法的性能相关参数上做文章。

（1）单分区 LOAD 优化建议

- 大多数情况下，在 LOAD 操作期间对现有索引进行增量维护（使用 INCREMENTAL 选项）而不是重建（使用 REBUILD 选项）更节省时间。
- 如果表上定义了很多索引，增加 SORTHEAP 和 SHEAPTHRES_SHR 的大小，增加临时表空间使用的缓冲池大小，可提高索引建立速度。
- 如果排序溢出过多，可考虑使用 INDEXING MODE DEFERRED 选项，以后再找机会分批重建索引。
- LOAD 在 INSERT 方式和 REPLACE 方式下运行得一样好，但是如果使用了 REBUILD 选项，用 REPLACE 方式导入比用 INSERT 方式导入更快。
- 增大实用程序堆（UTIL_HEAP_SZ）的大小，可以提升 LOAD 的速度。由于 DATA BUFFER 代表 LOAD 所需的缓存大小，这部分缓存是从实用程序堆（Utility Heap）里分配的，如果实用程序堆的尺寸大，那么 DATA BUFFER 就能获得足够的内存资源。
- 调整 DISK_PARALLELISM 和 CPU_PARALLELISM 的设置，可以提升 LOAD 的并行处理能力。DISK_PARALLELISM 代表处理 I/O 的进程数量，CPU_PARALLELISM 代

表处理格式化和解析等任务的进程数量。如果不指定这些值，LOAD 将尝试选择一个最优组合。

- 增大 SAVECOUNT 的值。如果 SAVECOUNT 比较小，会导致数据经常回写磁盘，产生大量 I/O。一般来说，最好还是把 SAVECOUNT 设得大一点，毕竟使用 LOAD 不用像使用 IMPORT 一样担心日志空间满的问题。
- 使用 FASTPARSE 可以提高 LOAD 速度。使用 FASTPARSE 文件类型修饰符来减少对导入数据的合法性检查，通常性能可以提高 10%~20%。但使用此选项，必须人为保证导入数据的合法性（只能针对 DEL 文件）。

（2）多分区 LOAD 优化建议

在多分区环境下，执行 LOAD 虽然和在单分区环境下执行有很多相似点，但是在分布数据方面存在着比较大的差异。为了方便广大读者，在这里分享一下多分区环境下使用 LOAD 操作的最佳实践。

- 在使用 LOAD 导入大批量数据之前，首先用少量数据进行测试，确保 LOAD 的各种选项设置正确。
- 如果想按导入文件中的顺序导入，就只能指定一个分区作为分布数据的分区，因为并行分布会给导入顺序带来不确定性。在默认情况下，如果不指定 anyorder，LOAD 只选择一个分区作为分布数据的分区。
- 如果需要导入 LOB 数据，那么要确保存放 LOB 数据的路径对于所有参与导入的数据库分区都是可见的。
- 可以将 ISOLATE_PART_ERRS 设置成 SETUP_AND_LOAD_ERRS 或者 SETUP_ERRS_ONLY 来防止某些意外错误中止整个 LOAD 导入过程，从而让没有出错的分区继续执行。
- 尽量让 PARTITIONING_DBPARTNUMS 指定的分区和 OUTPUT_DBPARTNUMS 指定的分区无交集，这样可以把分布数据和导入数据的任务分配在不同的分区上。同时，应该选择在既没有分布工作，也没有导入工作存在的分区上执行 LOAD 命令，从而减少 CPU 的使用冲突，以获得更好的导入性能。
- 多分区的统计信息收集只在一个分区进行，其他分区是假设数据分布均匀的前提下预测出来的。在 LOAD 过程中我们可以设定 RUN_STAT_DBPARTNUM 来决定在哪个分区上收集统计信息。
- 尽量使用 MESSAGES 选项保留分布和导入时产生的消息文件，便于在 LOAD 结束后用于问题追踪。

7.3.4　LOAD FROM CURSOR 轻松移动数据

在数据量很大的情况下，使用 EXPORT 实用程序生成数据交换文件常常要花费很长时间。另外，在把数据导出和导入数据库时，必须考虑不同的数据库编码页。

可以使用 LOAD FROM CURSOR 命令避免这些问题。当指定 FROM CURSOR 选项时，LOAD 直接把一个 SQL 查询的结果集做为数据装载操作的来源，这样就不需要生成临时数据交换文件。

可以在命令行上执行 LOAD FROM CURSOR，也可以通过使用 DB2 的 ADMIN_CMD 存储过程在应用程序或存储过程中执行它。例如，下面的 LOAD FROM CURSOR 命令把 SALES 表中的所有行复制到 SALES_TMP 表中，其中 load_sales_tmp.msg 为消息文件：

```
=>DECLARE C1 CURSOR FOR SELECT * FROM FECHNER.SALES
DB20000I  The SQL command completed successfully.

=>LOAD FROM C1 OF CURSOR MESSAGES load_sales_tmp.msg INSERT INTO
FECHNER.SALES_TMP NONRECOVERABLE

Number of rows read        = 41
Number of rows skipped      = 0
Number of rows loaded      = 41
Number of rows rejected     = 0
Number of rows deleted      = 0
Number of rows committed    = 41
```

7.3.5　INGEST——性能和可用性间平衡的使者

INGEST 实用程序又称为连续数据插入或 CDI，是 DB2 V10 引入的一个高速的客户端实用程序，它可以将数据以流的形式从文件和管道中以较高的速度插入到 DB2 目标表中。同时 INGEST 在移动大量实时数据的时候不需要像 LOAD 一样锁定目标表，它使用行锁定，因此对访问相同表的其他用户的影响较少，在性能和可用性之间做到了极好的平衡。

INGEST 不需要锁定目标表。它支持使用下列 SQL 语句：INSERT、UPDATE、MERGE、REPLACE 和 DELETE 来对表执行 DML 操作。它还支持使用 SQL 表达式来根据多个数据字段构建各个列值。

INGEST 支持 ESE、PureScale 和 DPF 环境。它与数据库引擎使用相同的标准化外部接口，是个多线程工具，速度快，效率高。

INGEST 还可以按执行的时间间隔或者写入的行数来确定落实事务的频率。可以使用 commit_count 参数来设置每写入多少行数据落实一次事务，或者使用 commit_period 参数来设置插入过程中两次事务落实的时间间隔。缺省情况下 commit_period 值为 1，即 INGEST 每 1 秒钟落实一次事务。如果同时设置了 commit_count 和 commit_period,，那么这两个参数同时起作用，即如果它已写入指定行数或指定秒数内没有落实操作，那么它会发出落实指令。通过 INGEST SET 命令，可以来调整 commit_count 参数为每插入 1000 行落实一次事务：

```
=>ingest set commit_count 1000
DB20000I  The INGEST SET command completed successfully
```

INGEST 可以将已拒绝的记录复制到文件（通过 DUMPFILE 参数指定）或表中（通过 EXCEPTION TABLE 参数指定），也可以直接废弃这些记录。

当 INGEST 意外中断时，还可以重新启动和恢复。缺省情况下，将从最后一个落实点重新启动所有 INGEST 命令。另外，如果设置了 retry_count 参数，那么 INGEST 实用程序也会尝试从某些错误中恢复。

INGEST 命令支持的输入数据格式和插入的目标对象如表 7-3 所示：

表 7-3　INGEST 支持的文件格式和目标对象类型

输入格式	目标对象
定界文本； 位置文本和二进制文件； 各种顺序和格式的列	常规表； 昵称； 多维集群（MDC）； 分区表； 由用户维护的物化查询表（MQT）； 临时表； 可更新的视图

单个 INGEST 命令将经历三个主要阶段：

1. 传输

传输程序从数据源中读取记录，并将记录放置在格式化程序队列中。对于 INSERT 和 MERGE 操作，每个输入源都有一个传输程序线程（例如，每个输入文件有一个线程）。对于 UPDATE 和 DELETE 操作，只有一个传输程序线程。

2. 格式化

格式化程序会解析每个记录，将数据转换成 DB2 数据库系统所需要的格式，并将每个已格式化的记录放置在该记录的分区上的其中一个清空程序队列中。格式化程序线程数由 num_formatters 配置参数指定。该参数的缺省值为逻辑 CPU 数/2。

比较：物理 CPU 个数、CPU 内核数和逻辑 CPU 个数

- 物理 CPU 个数：实际服务器中插槽上的 CPU 个数。
- CPU 内核数：物理 CPU 个数与每颗物理 CPU 的内核数的乘积。
- 逻辑 CPU 个数：没有开启超线程的情况下，逻辑 CPU 数目等于 CPU 内核数；如果开启超线程的话，逻辑 CPU 个数是 CPU 内核数的两倍。

---

3. 清空

清空程序发出 SQL 语句以对 DB2 表执行操作。每个分区的清空程序数由 num_flushers_per_partition 配置参数指定，该参数的缺省值为 1 与（逻辑 CPU 数/2/分区数）之中的较大者。

在使用 Ingest 工具之前，需要首先创建 restart 信息表，有了这张表，ingest 工具就可以将相关的进度信息存放在这张表中，以便下次 restart 的时候使用。可以使用系统存储过程 sysinstallobjects 来创建 restart 信息表，其第一个参数为工具名称，在这里为“INGEST”，第二个参数为操作的行为，“C”表示创建，第三、四两个参数分别表示对象所在的表空间和模式，当使用 null 时表示使用默认值。执行完成后，可以使用 describe 命令来查看创建好的 restart 信息表 INGESTRESTART，由于这张表是 INGEST 命令内部使用，在此就不做过多解释。

```
=> call sysproc.sysinstallobjects('INGEST','C',null,null)

  Return Status = 0
=> describe table "SYSTOOLS.INGESTRESTART

              Data type  Data type Column
Column name     schema     name      Length Scale Nulls
--------------- --------- ---------- ---    ----- ------
JOBID          SYSIBM    VARCHAR    256       0 No
APPLICATIONID   SYSIBM    VARCHAR    256       0 No
FLUSHERID       SYSIBM    INTEGER      4       0 No
FLUSHERDISTID   SYSIBM    INTEGER      4       0 No
TRANSPORTERID   SYSIBM    INTEGER      4       0 No
BUFFERID        SYSIBM    BIGINT       8       0 No
BYTEPOS         SYSIBM    BIGINT       8       0 No
ROWSPROCESSED   SYSIBM    INTEGER      4       0 No

  8 record(s) selected.
```

下面通过几个简单的例子讲解一下 Ingest 的用法。我们使用 Sample 数据库中的 staff 表作为演示的例子，首先将 staff 表导出：

```
=>export to staff.del of del select * from staff
SQL3104N  The Export utility is beginning to export data to file "staff.del".
SQL3105N  The Export utility has finished exporting "35" rows.
Number of rows exported: 35
```

最简单的业务场景是对数据不加任何处理，直接导入到和原表结构相同的表中：

```
=>create table staff02 like staff
DB20000I  The SQL command completed successfully.
=>INGEST FROM FILE staff.del FORMAT DELIMITED INSERT INTO staff02
SQL2979I  The ingest utility is starting at "12/10/2012 23:58:52.783010".
SQL2914I  The ingest utility has started the following ingest job:
"DB21001:20121210.235852.783010:00003:00009".

Number of rows read        = 35
Number of rows inserted     = 35
Number of rows rejected     = 0

SQL2980I  The ingest utility completed successfully at timestamp "12/10/2012
23:58:55.669288"
```

INGEST 支持在导入数据的同时，通过相应的运算，将运算结果导入到新增加的列中。下面这个例子先计算 SARLARY 和 COMM 的和，随后将结果插入到新增加的 INCOME 列中：

```
=> create table staff03 like staff
DB20000I  The SQL command completed successfully.
=>alter table staff03 add column income decimal(9,2)
DB20000I  The SQL command completed successfully.
=>INGEST FROM FILE staff.del FORMAT DELIMITED ($ID SMALLINT EXTERNAL,$NAME
CHAR(9),$DEPT SMALLINT EXTERNAL,$JOB CHAR(5),$YEARS SMALLINT EXTERNAL,$SALARY DECIMAL(7,2)
EXTERNAL,$COMM DECIMAL(7,2) EXTERNAL) INSERT INTO
staff03(ID,NAME,DEPT,JOB,YEARS,SALARY,COMM,INCOME)values($ID,$NAME,$DEPT,$JOB,$YEARS,
```

```
$SALARY,$COMM,NVL($SALARY,0)+NVL($COMM,0))
    SQL2979I  The ingest utility is starting at "12/12/2012 23:08:04.555699".
    SQL2914I  The ingest utility has started the following ingest job:
    "DB21001:20121212.230804.555699:00003:00010".

    Number of rows read        = 35
    Number of rows inserted     = 35
    Number of rows rejected     = 0

    SQL2980I  The ingest utility completed successfully at timestamp "12/12/2012
    23:08:07.084783"
```

INGEST 还支持在导入数据时更新匹配的行。下面的例子中，在导入数据时，如果存在和 ID 匹配的行，就用导入数据行中的 SALARY 值更新该 ID 对应的 SALARY 值，如果不存在匹配的行，则忽略该行：

```
    => INGEST FROM FILE staff.del FORMAT DELIMITED ($ID SMALLINT EXTERNAL,$NAME
CHAR(9),$DEPT SMALLINT EXTERNAL,$JOB CHAR(5),$YEARS SMALLINT EXTERNAL,$SALARY DECIMAL(7,2)
EXTERNAL,$COMM DECIMAL(7,2) EXTERNAL) UPDATE STAFF03 set (SALARY)=($SALARY) WHERE ID=$ID
    SQL2903W  Configuration parameter "NUM_FLUSHERS_PER_PARTITION" has been
    automatically adjusted to the following value: "1". Reason code= "12"
    SQL2979I  The ingest utility is starting at "12/12/2012 23:10:53.589666".
    SQL2914I  The ingest utility has started the following ingest job:
    "DB21001:20121212.231053.589666:00003:00010".

    Number of rows read        = 35
    Number of rows updated      = 35
    Number of rows rejected     = 0

    SQL2902I  The ingest utility completed at timestamp "12/12/2012
    23:10:54.691003". Number of errors: "0". Number of warnings: "1".
```

还可以在导入数据时删除匹配的行。下面在插入数据时，如果不存在 ID 匹配的行，就忽略该行，否则就删除原表中与 ID 匹配的行：

```
    => INGEST FROM FILE staff.del FORMAT DELIMITED ($ID SMALLINT EXTERNAL,$NAME
CHAR(9),$DEPT SMALLINT EXTERNAL,$JOB CHAR(5),$YEARS SMALLINT EXTERNAL,$SALARY DECIMAL(7,2)
EXTERNAL,$COMM DECIMAL(7,2) EXTERNAL)DELETE FROM STAFF03 WHERE ID=$ID
    SQL2903W  Configuration parameter "NUM_FLUSHERS_PER_PARTITION" has been
    automatically adjusted to the following value: "1". Reason code= "12"
    SQL2979I  The ingest utility is starting at "12/12/2012 23:12:31.517123".
    SQL2914I  The ingest utility has started the following ingest job:
    "DB21001:20121212.231231.517123:00003:00010".

    Number of rows read        = 35
    Number of rows deleted      = 35
    Number of rows rejected     = 0

    SQL2902I  The ingest utility completed at timestamp "12/12/2012
    23:12:32.475787". Number of errors: "0". Number of warnings: "1".
```

7.3.6 使用 db2move 在数据库间移动数据

db2move 是一个在数据库之间批量移动大量数据的工具。它支持的选项包括 EXPORT、IMORT、LOAD 和 COPY。其中 EXPORT、IMORT 和 LOAD 在与 db2move 结合的时候，可以同时对多个表（比如一个模式下面的所有表）进行导入导出。COPY 选项用来把一个或多个模式中的所有表和数据复制到另一个数据库中。听起来很有趣，我们来看一下 db2move 的使用方法。

```
    db2move dbname action options
```

这里的 action 指的是 EXPORT、IMORT、LOAD 和 COPY 其中之一。

- EXPORT：根据 options 中指定的过滤条件导出表，如果不指定 options，则导出数据库中所有表。导出的表信息存在一个叫做 db2move.lst 的文件中。
- IMPORT：导入 db2move.lst 中列出的表数据，IMPORT 选项由 options 中的-io 决定，包括 INSERT、INSERT_UPDATE、REPLACE、CREATE 和 REPLACE_CREATE，默认是 REPLACE_CREATE。
- LOAD：导入 db2move.lst 中列出的表数据，LOAD 选项由 options 中的-lo 决定，包括 REPLACE 和 INSERT，默认是 INSERT。
- COPY：需要借用 SYSTOOLSPACE 表空间，把一个或多个模式中的表数据复制到另一个数据库中，目标数据库必须是本地数据库。

命令中的 options 配合 action 一起使用，决定了导入或导出的方式和范围等信息，结合 COPY 选项，使用 db2move 把一个模式下的所有表从一个数据库复制到另一个数据库。

```
    [db2inst1@db2 ~]$db2move sample1 COPY -sn db2lde -co TARGET_DB sample2 USER db2lde
USING 123456 MODE DDL_ONLY

    Application code page not determined, using ANSI codepage 819

    [Detected DB2DBDFT env variable as "LDE", disabling for duration of utility.]

    *****  DB2MOVE  *****

    Action:   COPY

    Start time:   Fri Apr 29 23: 42: 19 2011
    All schema names matching:   DB2LDE;

    Connecting to database SAMPLE1 ... successful!  Server :  DB2 Common Server V9.5.3

    Copy schema DB2LDE to DB2LDE on the target database SAMPLE2

    Create DMT :   "SYSTOOLS"."DMT_4dbadc6218ce9"


    db2move finished successfully

    Files generated:
    -----------------
```

```
COPYSCHEMA.20110429234219.msg

Please delete these files when they are no longer needed.

[Re-enabling DB2DBDFT env variable to "LDE" on completion of utility.]
```

执行了这个例子的命令后，数据库 sample1 中凡是模式名字为 db2lde 的表将会被复制到目标数据库 sample2 中。其中，DDL_ONLY 选项表示只在目标数据库 sample2 建立表而不导入数据。当然还可以用 LOAD_ONLY 和 DDL_AND_LOAD，其中 LOAD_ONLY 表示只导入数据，DDL_AND_LOAD 表示建表并导入数据。

如果需要改变表所在的表空间，可以加入如下代码所示的 TABLESPACE_MAP 选项，这样原来在 TS1 表空间的表在复制时被放到 TS2 表空间，原来在 TS2 表空间的表被放到 TS3 表空间，SYS_ANY 表示其他表空间由数据库选择映射关系。

```
TABLESPACE_MAP ((TS1,TS2),(TS2,TS3),SYS_ANY)
```

除了原封不动的搬运，还可以在导入过程中进行模式映射。通过在 db2move 命令中使用 SCHEMA_MAP 选项在下例中将模式 source_schema1 映射为 target_schema1，source_schema2 映射为 target_schema2：

```
SCHEMA_MAP ((source_schema1,target_schema1),(source_schema2,target_schema2))
```

前面讲述了 db2move 的基本用法，但是在实战中，还需要设置正确的系统参数和数据库参数，下面就是由于系统参数没有正确设置导致命令执行失败的一个案例。

客户使用 DB2 V9.7.6，运行在 HP-UX 平台，使用 db2move mydb load -lo replace 命令进行数据库迁移。试验了很多次，当迁移了 10%左右的表时就会出错，检查 db2diag.log，发现下面的错误：

```
2012-11-27-22.03.59.040863+480 E4604733A800       LEVEL: Error (OS)
PID     : 11563              TID  : 11         PROC : db2sysc 0
INSTANCE: db2inst1            NODE : 000         DB   : NCXH
APPHDL  : 0-68               APPID: *LOCAL.db2inst1.121127135620
AUTHID  : DB2INST1
EDUID   : 11                 EDUNAME: db2agent (NCXH) 0
FUNCTION: DB2 UDB, oper system services, sqlowqueInternal, probe:40
MESSAGE : ZRC=0x870F0041=-2029060031=SQLO_QUE_NOT_SENT "Message Not Sent"
          DIA8557C No message was sent using the message queue.
CALLED  : OS, -, msgsnd
OSERR   : EAGAIN (11) "Resource temporarily unavailable"
DATA #1 : system V message queue identifier., PD_TYPE_SYSV_QUEUE_ID, 4 bytes
0x06C0004B
DATA #2 : Pointer, 8 bytes
0x9ffffffebfea670
DATA #3 : unsigned integer, 8 bytes
```

通过分析错误日志，可以看出是由于系统消息队列参数值过小，使用 db2osconf 对 HP-UX 内核参数进行推荐，但是给出有关消息队列的推荐值明显偏小，msgmnb、msgtql 和 msgmni 明显偏小，无法满足需要。

```
****************** System Resources **********************
Physical RAM ==> 137259171840
Number of CPUS ==> 16
System Scale Factor ==> 8192.00

* Recommended HP-UX Kernel Parameters Based on this System *

nproc ==> 8208
maxuprc ==> 2736
max_thread_proc ==> 32767
maxfiles ==> 4096
maxfiles_lim ==> 65535
nflocks ==> 16384
ninode ==> 11469
msgmnb ==> 65535
msgtql ==> 6554
msgmni ==> 6554
semmns ==> 16385
semmni ==> 8192
semmnu ==> 8192
shmmax ==> 123533254656
shmmni ==> 8192
maxdsiz_64bit ==> 1900000000
```

通过手工增大消息队列参数，将这三个参数值设置为：

```
msgmnb (max # of bytes on a message queue, i.e max-msg-qbytes in Solaris 10+) = 2097152
msgmni (max # of message queues, i.e max-msg-ids in Solaris 10+) = 32767
msgtql (max # of message on all message queues, i.e max-msg-messages in Solaris 10+)
= 262144
```

重新运行 db2move mydb load -lo replace，成功加载数据。

7.3.7 db2relocatedb——DB2 移形换位

DB2 数据库有大量的内部元数据，这些数据描述了用户数据存放位置、用户数据属于谁、以及如何引用它们。元数据总是被 DB2 隐藏在“幕后”，用户和 DBA 都不会直接接触它。

例如，DBA 需要修改数据库的名称，通常可以使用 DB2 的备份和恢复命令来完成。先备份数据库，然后用新的名字恢复。备份操作可以在线完成，但恢复操作是离线的，而且其间数据库不能使用，如果有高可用性要求，那么这种情况是不可接受的。

另外一个例子是改变表空间容器的位置，通常的做法是通过备份和重定向恢复完成。如果表空间是 DMS，那么还有一种选择，那就是首先使用 ALTER TABLESPACE 语句向新的位置添加容器，然后再从原来位置删除容器。这种方法的副作用是操作必须按顺序进行，操作之后会出现表空间重平衡。

上面的例子中，其实都是要求修改 DB2 的元数据，但是，备份和恢复要做到这一点代价太大，难以满足实际需要。本节中将要讲到的 db2relocatedb 工具在这些场景中却大有用武之地。

db2relocatedb 并不修改数据库的实际物理位置，而是通过修改 DB2 元数据的方式来重新指定数据库的物理位置，比如容器位置和日志目录位置，也可以用来更改数据库的名称。

使用 db2relocatedb 之前，需要停止 db2 实例，其命令格式很简单，只需要运行：

```
    db2relocatedb -f relocate.cfg
```

关键的内容在 relocate.cfg 配置文件里，可以在该文件中定义需要变更的内容，包括数据库的名字（DB_NAME）、数据库的路径（DB_PATH）、实例（INSTANCE）、分区节点的编号（NODENUM）、在线日志路径（LOG_DIR）、容器的位置（CONT_PATH）、自动存储管理的存储路径（STORAGE_PATH）、归档失败路径（FAILARCHIVE_PATH）、归档方法（LOGARCHMETH1，LOGARCHMETH2）、镜像在线日志路径（MIRRORLOG_PATH）、前滚恢复时归档日志所在位置（OVERFLOWLOG_PATH）。

接下来，通过一个实战案例来说明 db2relocatedb 的用法。

一个客户重启 Linux 服务器后，db2 实例可以启动，但是无法成功连接，通过检查 db2diag.log，发现了下面的错误：

```
    2012-11-06-13.43.03.157979+480 I49705715E481      LEVEL: Error
    PID     : 910                 TID  : 47931096820032PROC : db2sysc 0
    INSTANCE: db2inst1             NODE : 000          DB   : YXDB
    APPHDL  : 0-19                APPID: 10.137.249.101.2938.121106054304
    AUTHID  : DB2INST1
    EDUID   : 31                  EDUNAME: db2agent (YXDB) 0
    FUNCTION: DB2 UDB, base sys utilities, sqle_GetRelDBLockForInstance, probe:25
    MESSAGE : /arch/db2log/db2inst1/NODE0000/SQL00001/SQLINSLK
    2012-11-06-13.43.03.157979+480 I49706197E572      LEVEL: Severe
    PID     : 910                 TID  : 47931096820032PROC : db2sysc 0
    INSTANCE: db2inst1             NODE : 000          DB   : YXDB
    APPHDL  : 0-19                APPID: 10.137.249.101.2938.121106054304
    AUTHID  : DB2INST1
    EDUID   : 31                  EDUNAME: db2agent (YXDB) 0
    FUNCTION: DB2 UDB, base sys utilities, sqeLocalDatabase::FirstConnect, probe:5
    RETCODE : ZRC=0x870F0016=-2029060074=SQLO_SHAR "File sharing violation."
             DIA8519C A file sharing violation has occurred, filename was "".
```

这种错误通常是由于 db2 相关进程，例如 db2cos 没有释放加在控制文件上的锁导致的。常规的解决办法是使用 db2_kill 杀掉其他进程，以及用 ipclean 清理 IPC 资源。但是现场经过各种尝试，甚至也试了重启机器，这个问题都无法解决。

后来经过检查，发现客户所使用数据库 mydb 的 DB PATH（/arch/db2log）挂载在 NFS 文件系统上，所以尝试着用 db2relocatedb 命令将 DB PATH 从/arch/db2log 移动到盘阵/data1/dbpath，最终解决了这个问题。

整个处理步骤如下：

（1）检查原来的建库语句，发现 DB PATH 创建在 NFS 文件系统/arch/db2log 上：

```
    create database mydb automatic storage yes on /data1, /data2, /data3 dbpath on
/arch/db2log using codeset UTF-8 territory CN pagesize 32
```

（2）复制/arch/db2log 下所有文件到新的路径/data1/dbpath：

```
cp -rf /arch/db2log/*   /data1/dbpath
```

（3）编辑移动文件 rel.cfg，内容如下：

```
DB_NAME=mydb
DB_PATH=/arch/db2log,/data1/dbpath
INSTANCE=db2inst1
```

（4）使用 db2relocatedb 命令修改 DB PATH：

```
db2relocatedb -f rel.cfg
Files and control structures were changed successfully.
Database was catalogued successfully.
DBT1000I  The tool completed successfully.
```

（5）成功连接：

```
        db2 connect to mydb
Database Connection Information
    Database server        = DB2/LINUX 9.7.6
    SQL authorization ID   = DB2INST1
    Local database alias   = mydb
```

注意：db2relocatedb 的使用

db2relocatedb 是一种离线工具，绝不能在数据库开启时运行它。

下面是使用 db2relocatedb 工具的最佳实践：

（1）db2relocatedb 对数据库文件和控制结构的修改不记入日志，这些修改是不可修复的。因此，强烈建议在对数据库使用该工具之后，进行完整的备份。

（2）如果要进行重要的修改或者进行用户认为可能难以逆转的修改，最好考虑在修改之前进行备份。

（3）虽然 db2relocatedb 工具可以改变表空间容器的位置，但不能用它来改变容器的数量和大小，唯一办法是使用重定向恢复。

（4）将所有数据库从一个路径移动到另一个路径是完全有可能的，只需为每个数据库创建配置文件并运行 db2relocatedb 即可。

如果移动了数据库，但是忘记移动容器，或者忘记在配置文件中添加上相应的条目，仍然可以连接到数据库。这时可以使用 LIST TABLESPACES SHOW DETAIL 查看所有表空间的状态，确保没有一个表空间处于 OFFLINE 状态。如果处于 OFFLINE 状态，应该关闭数据库，改正出现的问题，然后再重新运行该工具（该工具拥有足够的智能了解自己已经完成了该工作的一部分，因此第二次运行不会造成任何损害）。

7.3.8 存储过程——内部数据挪移的利器

从 DB2 V9.7 开始，提供了两个系统存储过程 ADMIN_MOVE_TABLE 和 ADMIN_COPY_SCHEMA，它们都位于 SYSPROC 模式下面。前者可以将表从一个表空间移动到另一个表空间，包括表数据，所有的表选项、索引和视图也都会被移动。在移动过程中，表处于在线可访问状态（只在一个短暂阶段处于只读状态）。它也可以实现离线表移动，而且速度明显快于在线表

移动。后者有点像在 db2move 中使用的 COPY 选项，可以用来复制某个 SCHEMA 中的全部对象以及数据，但是它只支持同一数据库内复制。

1. 使用 ADMIN_MOVE_TABLE 在数据库内移动表

如图 7-6 所示，展示了在线表移动过程中各个表的作用和关系。

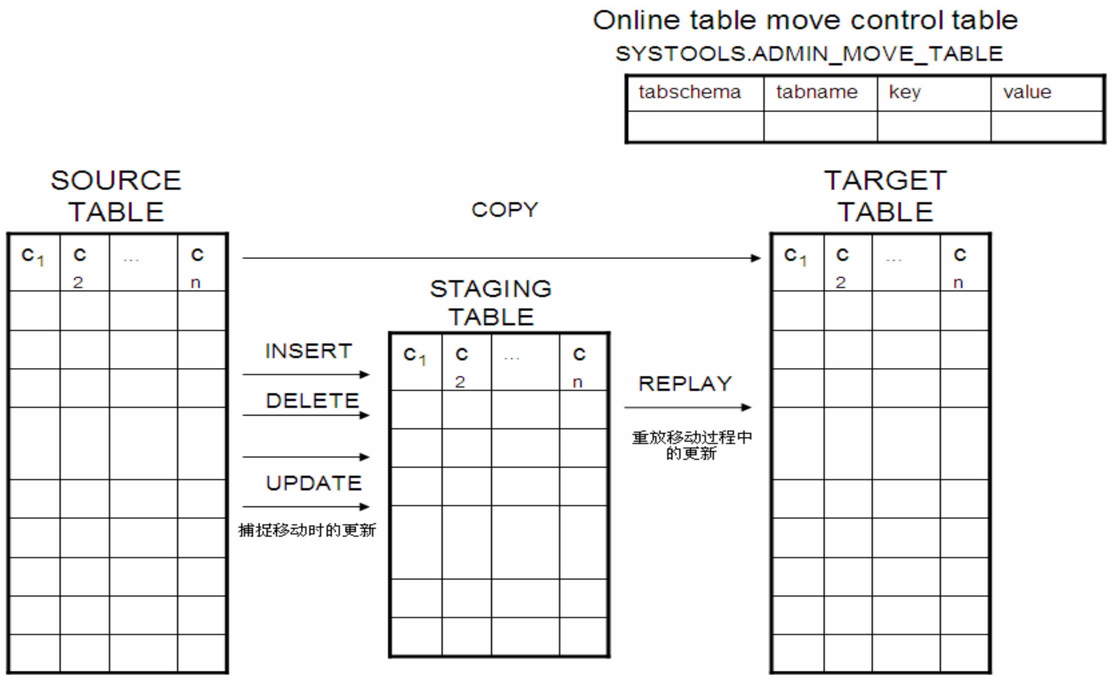


图 7-6　在线表移动过程示意图

从图 7-6 中可以看到，主要涉及 4 个重要的表：

- 源表（Source table）：需要被移动的表。
- 目标表（Target table）：在目标位置建立的表，表定义与源表一样，用来接收源表的数据，在所有数据移动完成后，目标表会被重命名为源表的表名。
- 阶段表（Staging table）：用来记录移动过程中其他事务对表的修改，移动完成后这些修改会在目标表重放，结束后阶段表会被删除。
- 移动控制表（SYSTOOLS.ADMIN_MOVE_TABLE）：记录了表移动时的进度和状态信息，这个表存放于 SYSTOOLSPACE 表空间中。

在线表移动的方法主要分为两种：一种方法基本上属于全自动，只允许改变部分目标表的定义，不需要在移动前手动建立目标表。第二种方法提供了更灵活的控制和选择，但需要事先建立好目标表。通常第一种方法是比较常用的方法，你只需要指定目标数据表空间、目标索引表空间和目标 LOB 表空间，其他参数可以置空。

通过下面的实例来学习如何把一个表从一个表空间移动到另一个表空间。

```
    => db2 "select char(tabname,8) as table,char(tbspace,8) as tablespace from
syscat.tables where tabname='STAFF'"
    TABLE    TABLESPACE
    -------- ----------
```

```
    STAFF    TS1
    1 record(s) selected.

    => db2 "CALL SYSPROC.ADMIN_MOVE_TABLE('DB2LR1', 'STAFF', 'TS2','TS2','TS2','', '',
'', '','','MOVE')"
      Result set 1
      --------------
      KEY                            VALUE
      --------------------------------
-------------------------------------------------------------------------------------
------------------------------------------
      AUTHID                          DB2LR1
      CLEANUP_END                      2011-04-30-07.01.46.564862
      CLEANUP_START                    2011-04-30-07.01.46.544574
      COPY_END                        2011-04-30-07.01.45.823364
      COPY_OPTS                       ARRAY_INSERT,NON_CLUSTER
      COPY_START                      2011-04-30-07.01.45.729002
      COPY_TOTAL_ROWS                   35
      INDEXNAME
      INDEXSCHEMA
      INIT_END                        2011-04-30-07.01.45.714766
      INIT_START                      2011-04-30-07.01.45.195087
      REPLAY_END                      2011-04-30-07.01.46.434644
      REPLAY_START                     2011-04-30-07.01.45.823836
      REPLAY_TOTAL_ROWS                 0
      REPLAY_TOTAL_TIME                 0
      STATUS                          COMPLETE
      SWAP_END                        2011-04-30-07.01.46.518082
      SWAP_RETRIES                     0
      SWAP_START                      2011-04-30-07.01.46.489645
      VERSION                         09.07.0004

    20 record(s) selected.

      Return Status = 0
    => db2 "select char(tabname,8) as table,char(tbspace,8) as tablespace from
syscat.tables where tabname='STAFF'"

    TABLE    TABLESPACE
    -------- ----------
    STAFF    TS2
      1 record(s) selected.
```

从例子中可以看到表 STAFF 被成功地从表空间 TS1 移动到表空间 TS2，其索引以及 LOB 数据也都被移动到表空间 TS2 中。在例子中我们用了 MOVE 选项，执行 INIT、COPY、REPLAY 和 SWAP 步骤，这是一种一气呵成的办法。

除了 MOVE 以外，还可以分步实现数据移动，这些步骤包括 INIT、COPY、REPLAY、VERIFY、SWAP、CLEANUP 和 CANCEL。

- INIT：初始化目标表、staging 表和触发器。
- COPY：从源表向目标表复制数据，复制过程中，对源表的操作将记录在 staging 表中。

- REPLAY：将 staging 表中记录的操作在目标表上重放。
- VERIFY：可选，检查源表和目标表中的内容是否完全一致。
- SWAP：源表下线，目标表上线。
- CLEANUP：清理现场，例如删除 staging 表；删除建立在源表上的非唯一性索引和触发器；如果 KEEP 选项没有指定的话，删除源表。
- CANCEL：取消数据移动。

下面是使用 ADMIN_MOVE_TABLE 的先决条件和限制。

首先需要注意的是，ADMIN_MOVE_TABLE 不支持具有外键约束的表，并且在执行前需要确保数据库里有 SYSTOOLSPACE 表空间。

其次在移动过程中对源表进行的某些操作会使源表和目标表数据不一致，这些操作包括：

- TRUNCATE TABLE （没有使用 restrict when delete triggers 选项）。
- IMPORT…REPLACE INTO…（用替换方式进行导入）。
- LOAD TABLE。
- ALTER TABLE。
- REORG。

为了限制这些操作，DB2 使用了一个表级的状态标记，这个标记在 INIT 阶段设置，在 CLEANUP 或者 CANCEL 时解除。在数据移动期间，如果试图执行上述这些被限制的操作，那么这些操作会返回 SQL0668N（原因代码为 10）的错误信息。

通过查询移动控制表（SYSTOOLS.ADMIN_MOVE_TABLE）可以查看当前表移动的状态。例如下面的查询可以返回正在移动数据的表状态：

```
    [db2inst1@db2 ~]$db2 "SELECT varchar(tabschema,10) as schema, varchar(tabname,10) as
table, varchar(value,10) as status FROM  SYSTOOLS.ADMIN_MOVE_TABLE WHERE key = 'STATUS'
AND  VARCHAR(value,10) <> 'COMPLETE' WITH UR"

    SCHEMA     TABLE      STATUS
    ---------- ---------- ----------
    DB2LR1     SALES      COPY
    DB2LR1     PRODUCT    INIT

      2 record（s） selected.
```

从上例中可以看到有两个表正在做表移动，表 SALES 正处于 COPY 状态，而 PRODUCT 正处于 INIT 状态。

当用户有以下需求时，可以考虑使用 ADMIN_MOVE_TABLE：

（1）在 DB2 V9.7 对存储结构进行了调整，通过 ALTER TABLESPACE 语句与 REDUCE 选项配合使用，可随时回收未使用的存储空间，这简化了表空间的碎片管理。但是从以前版本升级上来的数据库的表空间还是使用过去的存储结构，这时我们可能需要使用在线表移动把已经存在的表移动到新建的可回收表空间，从而利用可回收表空间的好处。

（2）为了突破源表空间大小限制，可以把表从一个表空间移动到一个容量更大的表空间。比如把一个表从一个页大小为 16KB 的表空间移动到 32KB 页大小的表空间。

（3）把一个大表移动到另一个表空间，为目标表空间建立独立的缓冲池，从而提高性能。

（4）把表移动到不同类型的表空间中，比如将表从 SMS 管理表空间移动到 DMS 管理表空间。

（5）移动表从而降低高水位标记。

（6）把只读或者不经常更新的表移动到单独的表空间，从而提高增量备份的性能。

（7）把 LONG/LOB 数据移动到其他开启系统缓存的表空间，从而提高 LONG/LOB 数据的读取性能。

（8）不需要离线重组（Reorg）就可以实现对表的压缩。

（9）可以在线改变表的一些属性。例如启用或者撤销 MDC；启用或者撤销表分区（table-partition）；改变分区键；改变列的类型，比如从 LONG VARCHAR 变成 VARCHAR，或者从 LONG VARCHAR 变成 CLOB；增加或删除列；改变列的顺序等。

（10）在多分区环境下实现在线重新分布数据。

（11）DB2 V9.7 中引入了内嵌 LOB 特性，如果改变了内嵌 LOB 的长度定义，需要对表进行离线重组，使用 ADMIN_MOVE_TABLE 则可以实现在线转换，从而避免离线重组造成的表不可访问。

2. 使用 ADMIN_COPY_SCHEMA 在数据库内移动对象

使用 ADMIN_COPY_SCHEMA，必须具有 CREATE_SCHEMA 权限和所涉及对象的特权（比如建表的特权以及 EXECUTE 特权）。另外，ADMIN_COPY_SCHEMA 和 ADMIN_MOVE_TABLE 一样也需要使用 SYSTOOLSPACE 表空间。

使用 ADMIN_COPY_SCHEMA 的示例如下：

```
    [db2inst1@db2 ~]$db2 "CALL SYSPROC.ADMIN_COPY_SCHEMA('DB2LR1', 'LR1ADM', 'COPY', NULL, 
'', 'SYS_ANY', 'ERRORSCHEMA', 'ERRORNAME') "

      Value of output parameters
      --------------------------
      Parameter Name  :  ERRORTABSCHEMA
      Parameter Value :  -

      Parameter Name  :  ERRORTABNAME
      Parameter Value :  -

      Return Status = 0
```

上面的例子中，执行过程中使用了 COPY 参数，表示在建立目标表以后使用 LOAD 导入数据，行为与 NONRECOVERABLE 导入方式相同，最终，模式名为 DB2LR1 里的所有对象都被复制到了模式 LR1ADM 中。

关于参数设定方面，还可以使用 DDL 参数，复制过程中只建表，不导入数据，还有 COPY NO 参数，表示建表并使用 COPY NO 方式导入数据。另外，导入过程中可以将源表复制到不同的目标表空间，上例中使用 SYS_ANY，这表示数据库自己判断导入哪个表空间。

导入过程中的错误信息被记录在 ERRORTABNAME 表中，该表是由 ADMIN_COPY_SCHEMA 在 SYSTOOLSPACE 表空间里建立的。最后，操作完成以后最好做一个备份，防止前滚恢复导致表不可访问。

7.3.9　特殊对象的移动方式

众所周知，细节决定成败。在这节中我们就来学习一下如何处理 DB2 数据移动过程中的两种特殊对象。

1. IDENTITY 字段

在一个测试库中，我们使用 db2move 命令把所有表移动到另一个数据库过程中，发现大部分表数据能恢复，但有些却无法恢复并且提示 SQL3550W 错误。这是怎么回事呢？

在做数据移动的时候一定要记住，在 DB2 的表中有一种特殊字段，该字段的值是由系统分配的，这类特殊的字段叫 IDENTITY 字段。

```
create table MYTBL (c1 int,
                    c2 int generated always by identity,
                    c3 char(10))
```

在上面的定义中，可以看到表 MYTBL 中有一列 C2，它的值是在插入时由 IDENTITY 自动生成的，而 GENERATED ALWAYS 表明该字段只能由系统生成，不能由用户指定。对于整型的列，默认情况下 IDENTITY 的值从 1 开始每次加 1，当然也可以进一步定制 IDENTITY 的细节，规定它的初始值和递增值等详细信息。

IDENTITY 的具体使用并不是本节的重点，在这一节主要关注如何在数据移动时对这些特殊对象进行处理。先来看看如何向表 MYTBL 插入数据：

```
[db2inst1@db2 ~]$db2 insert into mytbl (c1, c3) values (1,'col1')
DB20000I  The SQL command completed successfully.

[db2inst1@db2 ~]$db2 insert into mytbl (c1, c3) values (2,'col2')
DB20000I  The SQL command completed successfully.

[db2inst1@db2 ~]$db2 select * from mytbl

C1          C2          C3
----------- ----------- ----------
          1           1 col1
          2           2 col2

  2 record(s) selected.
```

在上面的例子中，向表里插入了两行数据，在插入的时候并没有指定 C2 列的值，在插入之后可以看到表里 C2 列的值是自动生成的，从 1 开始每次加 1。

如果在插入时指定 C2 列的值会是什么结果呢？请看如下例子：

```
[db2inst1@db2 ~]$db2 insert into mytbl (c1, c2, c3) values (2,8,'col2')
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0798N  A value cannot be specified for column "C2", which is defined as
GENERATED ALWAYS.  SQLSTATE=428C9
```

在上面的插入语句中指定了 C2 列的值，在执行语句中会提示因为 C2 是指定了 GENERATED ALWAYS 选项的 IDENTITY 列，因此插入被拒绝了。如果在定义 IDENTITY 列时使用的是 GENERATED BY DEFAULT 选项，则允许插入时指定 C2 列的值。

既然插入受到 IDENTITY 列限制，对于导入来说也会受到同样的限制，特别是指定了 GENERATED ALWAYS 选项的 IDENTITY 列。接下来分别看看在进行 LOAD 导入时如何处理带有 GENERATED ALWAYS 选项的 IDENTITY 列。

在使用 LOAD 导入可以结合 identityoverride 选项进行导入，比如，

```
db2 "load from myfile.del of del modified by identityoverride insert into mytbl"
SQL3551W  The table contains at least one GENERATED ALWAYS column that the
utility will override.

SQL3109N  The utility is beginning to load data from file "myfile.del".

SQL3500W  The utility is beginning the "LOAD" phase at time "2011-12-13
15:52:14.635686".

SQL3519W  Begin Load Consistency Point. Input record count = "0".

SQL3520W  Load Consistency Point was successful.

SQL3110N  The utility has completed processing.  "2" rows were read from the
input file.

SQL3519W  Begin Load Consistency Point. Input record count = "2".

SQL3520W  Load Consistency Point was successful.

SQL3515W  The utility has finished the "LOAD" phase at time "2011-12-13
15:52:15.160134".


Number of rows read        = 2
Number of rows skipped      = 0
Number of rows loaded       = 2
Number of rows rejected      = 0
Number of rows deleted      = 0
Number of rows committed     = 2


[db2inst1@db2 ~]$db2 select * from mytbl

C1          C2          C3
----------- ----------- ----------
          1           1 col1
          2           2 col2

  2 record(s) selected.
```

在上面的例子中，C2 列的值是在导入时由导入文件指定的而不是自动生成的，identityoverride 还有一个作用是拒绝那些在导入文件中没有提供 IDENTITY 列值的行。关于上面提到的 3 个文件修饰符的具体含义，如表 7-4 所示：

表 7-4　导入 IDENTITY 字段时所采用的文件修饰符

文件修饰符	支持 IMPORT	支持 LOAD	描述
Identityignore	是	是	忽略输入文件中任何标识列的值，并为每一行生成一个新的标识值
Identitymissing	是	是	输入文件中不包含任何目标表中标识列的值
Identityoverride	否	是	在将数据导入到目标表中时，指定使用输入文件中标识列的值

这样数据不一致的问题就迎刃而解了。

另外，使用 LOAD，由于目标数据库的表总是新建的，如果想保证 IDENTITY 字段值能够连续，我们需要在目标数据库中手动更新目标表 IDENTITY 字段的值。例如通过 ALTER COLUMN 语句把 C2 列的起始值改成了 3，从而保证了目标数据库与源数据库的数值连续性。

```
ALTER TABLE mytbl ALTER COLUMN C2 RESTART WITH 3
```

2. 序列（SEQUENCE）

序列也是 DB2 中一类特殊的对象，它与 IDENTITY 字段有些类似，都可以自动生成顺序值，但序列是全局的，并不依赖于某个表的字段，因此可以实现全局范围内的顺序值生成。正因为这个值是全局的，我们在数据移动时对这个特殊对象更要十分注意。

比如，在一个测试库中，有一个序列 myseq，现在使用 db2move 命令将其移动到位于不同机器的数据库中，看起来一切正常。但是，运行一段时间后，才发现序列的号码出现了错误，这是怎么导致的呢？

在了解真相前，来看一个序列的定义：

```
CREATE SEQUENCE myseq AS BIGINT
     MINVALUE 1 MAXVALUE 65535
     START WITH 1 INCREMENT BY 1
     CACHE 20
     NO CYCLE
```

上面定义的序列 myseq 从 1 开始，每次递增 1，其最小值为 1，最大值为 65535，CACHE 20 代表序列每次生成 20 个值缓存在内存里，NO CYCLE 表示当序列生成到最大值以后不会循环复用以前的值了，从而保证了唯一性。

从序列的定义可以看出，序列是一个全局的对象，并不针对某个表，它会在内部保存当前序列的状态，一旦序列删除后重建，原来的状态就不复存在了。这也回答了之前提到的 db2move 后序列号码有错的问题：在进行数据移动过程中，序列只能在新数据库里重建，这样它的值又从起始值开始。

因此，在进行数据移动的时候，一定不要忘记根据源数据库的序列状态对新数据库中的序列状态进行更新。比如序列 myseq 在源数据库中的下一个生成值是 40，那可以在目标数据库发出如下命令：

```
ALTER SEQUENCE myseq RESTART WITH 40
```

执行了上述语句后，当在目标数据库中再次使用序列值的时候就可以看到当前序列从 40 开始了。

```
db2 values next value for myseq
1
--------------------
               40
1 record(s) selected.
```

7.4 备份恢复工具优化

在信息日趋发达的今天，数据显得尤其重要。如何保障数据的安全性呢？如何降低数据灾难事故发生时的损失呢？数据库备份与恢复作为数据安全最有利的保障手段的重要性就不言而喻了。而随着数据量的急剧增加，必须优化备份与恢复的整个过程，制定合理的备份与恢复策略，才能在有限的维护窗口完成备份与恢复，从而在保证数据安全的同时，将其对于生产系统的影响降到最低。

7.4.1 备份的基本原理与优化

数据库备份概念其实挺简单的：就是将数据库中所有数据存储为一个数据副本，供数据库出现问题时恢复使用。说白了，类似于我们将电话号码簿文件从 C 盘复制到 D 盘进行备份。

但是，与文件复制不同的是，我们并不能简单地在 DB2 上使用 copy 命令，而是需要通过 DB2 提供的备份工具（Backup Utility）来完成，具体的使用方法如下：

```
db2 backup db SAMPLE to /db2/db2lde
```

如图 7-7 所示，备份工具（Backup Utility）会读取数据库配置信息，所有的表空间结构和数据以及数据库历史记录文件，并将它们保存到数据库备份介质中；如果开启 Include Log 参数，数据库日志也会一同保存。

图 7-7 中，DB2 备份工具上有 4 个方块，分别是数据库配置文件、数据库表空间中的所有数据、数据库日志文件以及数据库历史记录文件。

中间的深黑色框代表数据库备份工具，即我们提到的 Backup 命令。

最下面的大框代表了最终保存的数据库备份介质，其中包括了上面读取的 4 个方块的信息。

那么，生成的数据库备份介质是什么样子呢？实际上，数据库备份介质是以文件的形式存储在磁盘上，如图 7-8 所示。

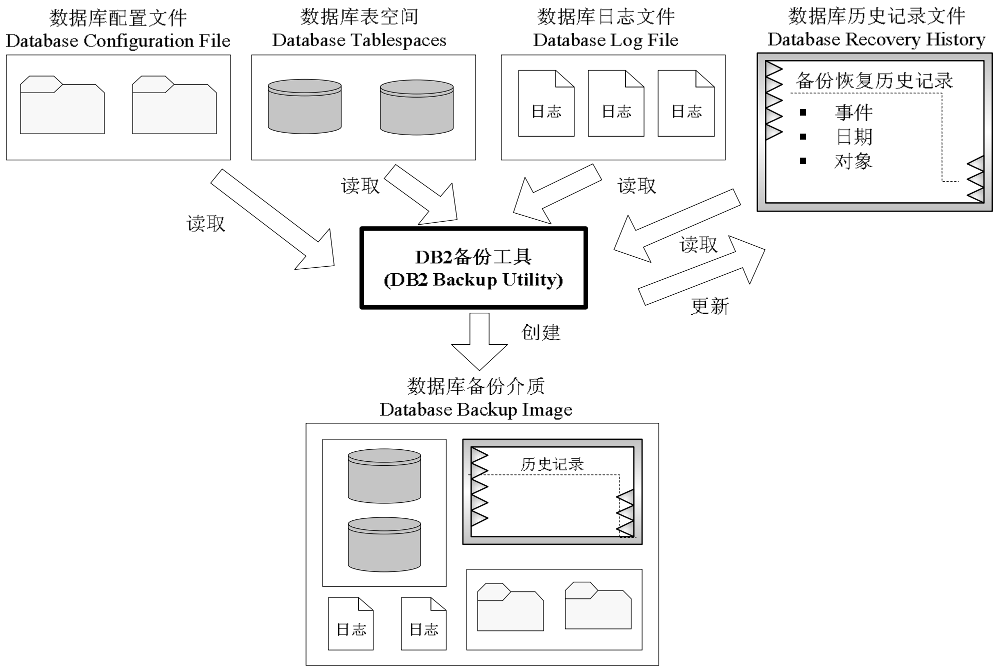


图 7-7　数据库备份介质生成过程

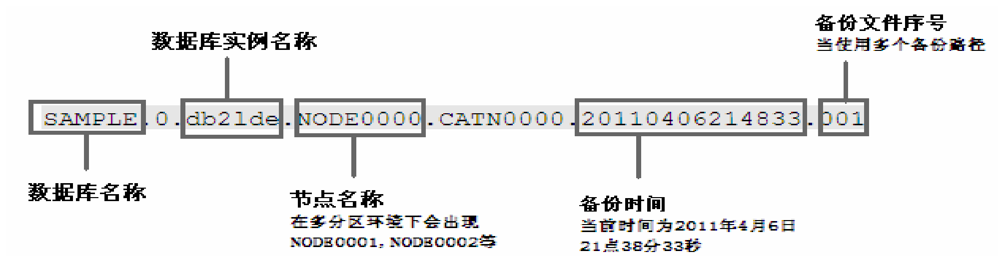


图 7-8　数据库备份介质文件名格式

在图 7-8 中，SAMPLE 是数据库名字，db2lde 是当前的数据库实例名字，NODE0000 为当前数据库节点名字。如果我们仅仅使用单分区数据库，那么名字总是 NODE0000，在多分区下，根据节点的多少会出现 NODE0001 或 NODE0002 等依次排列的编号。20110406214833 指的是备份的时间，其格式为 YYYYMMDDHHMMSS。末尾的 001 是备份介质的编号，例如当我们使用多个备份路径时，如 backup db SAMPLE to /db2_1，/db2_2 时，会产生 2 个备份介质，编号分别为 001 和 002。

数据库备份的原理虽然比较简单，但是备份过程会对生产系统产生一定影响，因此通常只能选择在相应的维护窗口进行，为了充分地利用系统资源，维护窗口的时间通常是有限的，因此必须制定合理的备份策略，快速而高效地备份。

1. 选择备份方式

根据业务系统能否中断，可以选择是进行离线备份还是在线备份，如果在维护窗口，允许业务中断，可以选择进行离线备份，离线备份时不允许业务访问数据库，但是备份速度较快。如果应用不允许中断，就必须选择在系统压力较少的时候进行在线备份（Online Backup）了。下面通过两个简单的实例来学习如何进行离线备份和在线备份。

首先切换到数据库管理员用户，当前为 db2lde，进行数据库离线备份：

```
db2 backup db SAMPLE to /db2/db2lde COMPRESS

Backup successful. The timestamp for this backup image is : 20110407064154
```

其中，SAMPLE 表示当前所备份的数据库名字，关键字 to 指定了备份介质的存储位置为 /db2/db2lde 目录，COMPRESS 指定对当前的数据库备份介质进行压缩。

接下来进行数据库在线备份，在线备份命令与离线备份命令的区别在于需要在备份命令中加入 online 关键字。与离线备份相同，SAMPLE 是备份的数据库名，关键字 to 指定了备份介质存储的位置，而 COMPRESS 参数指定启用压缩，实际命令如下：

```
db2 backup db SAMPLE online to /db2/db2lde COMPRESS
Backup successful. The timestamp for this backup image is : 20110727190638
```

> **注意：DB2 在线备份**
>
> 在线备份时，数据库必须处于归档模式下才能进行。

2. 选择备份对象

为了提高备份速度，可以根据业务特点选择合适的备份对象来减少需要备份的数据。

根据备份的对象可分为数据库备份和表空间备份（Tablespace Backup）。数据库备份是对整个数据库所有的数据所做的备份，而表空间备份则是针对某个或者某些经常变化而又比较重要的表空间所做的备份。如果数据库中只有部分表空间的数据经常变化且比较重要，而其他表空间的数据基本不变或者即使丢失，对于业务的影响也不大，这个时候就没有必要备份整个数据库，而只需要针对这些重要而经常变化的表空间进行表空间级别的备份，由于表空间级的备份较数据库备份数据量较少，因此其备份和恢复的时间也会相应缩短。

3. 选择备份规模

影响备份速度的另一个重要因素是备份规模，根据业务的需要和最小恢复时间的要求，合理地选择备份规模，可以有效地降低备份过程对于业务的影响。

根据备份的规模可以分为全备（Full Backup）和增量备份（Incremental Backup），完全备份是对数据库或表空间的所有数据所做的备份，而增量备份是对上一次备份以来变化的数据所做的备份，增量备份又可以分为累积增量备份（Cumulative Backup）和 Delta 备份（Delta Backup）。如图 7-9 所示，累积增量备份是包含从上一次全备以来的所有数据，中间可能有多次累积增量备份，而 Delta 备份只包含上一次备份以来变化的数据，如图 7-10 所示。

完全备份由于需要备份整个数据库或者表空间，数据量比较大，因此备份的时间较长，不过使用完全备份进行恢复时，其需要的恢复时间较短，而增量备份只需要备份部分数据，相比完全备份，其数据量较小，可以快速完成备份，但是恢复时，通常需要较长的恢复时间，为了同时满足较快的备份和最小恢复时间的要求，通常会采用完全备份和增量备份结合的方式，即根据业务的特点，选择每周或者每月进行一次完全备份，而每天进行 Delta 增量备份，每几天进行一次累积的增量备份方式。

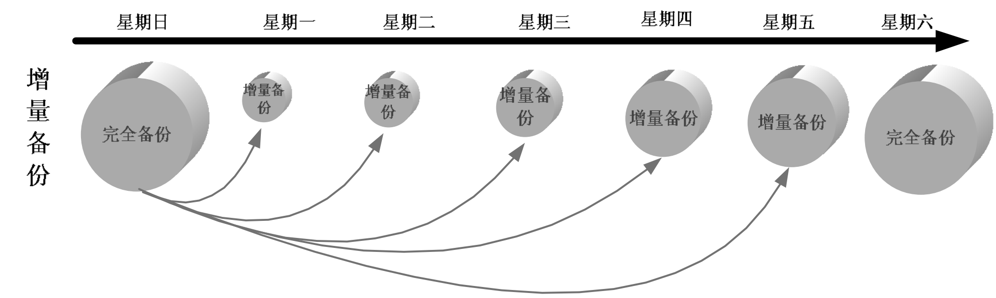


图 7-9 增量备份

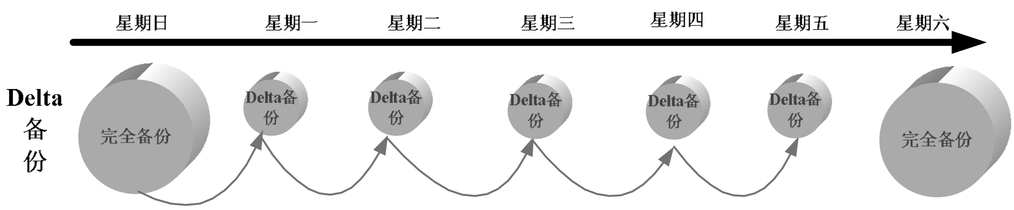


图 7-10 Delta 备份

> **小技巧：IBM 的 MergeBackup 工具很强大**
>
> 在一些特殊的情况下，由于业务量大，采用增量备份时，不能满足最小恢复时间的要求，而系统由于业务的压力又没有足够的维护窗口来进行完全备份，这个时候，就可以使用 IBM 提供的 MergeBackup 工具，它可以在不影响业务系统的基础上，合并旧的完全备份和其后的增量备份为新的完全备份，从而在不增加维护窗口的前提下，保证最小恢复时间的要求。

4. 选择备份参数

在制定正确的备份策略的基础上，还可以通过选择合适的备份参数来进一步优化备份的速度，以下参数可以用来调整备份性能：

```
BACKUP DATABASE <DB> TO dir | dev WITH num-buf BUFFERS
BUFFER buff-size PARALLELISM n COMPRESS
```

（1）To dir | dev

如果有多个目标路径被指定，如 dir1、dir2、dir3 等，这样会有多个 db2med（备份和恢复介质控制器）进程并行地向 dir1、dir2、dir3 等多个路径同时写入数据。

如果系统中有多个磁盘系统，可以使用多个目标路径分别指向这些磁盘系统，以充分利用多个磁盘系统的并行能力。

（2）WITH num-buf BUFFERS

每一个备份过程可以拥有一个或者多个内部缓存。将内部缓存的数目设置为 db2med 进程数目的 2 倍以上可以提升性能，因为这使得 db2med 进程向外部磁盘写入数据时无须等待缓存。

需要注意的是，（buff-size）×（num-buf）的值需要小于 UTIL_HEAP_SZ。UTIL_HEAP_SZ 指定了 DB2 中实用程序堆的大小，此参数决定备份、恢复和 LOAD 命令可以同时使用的最大

内存量。

（3）BUFFER buff-size

使用 BUFFER buff-size 参数可以指定内部缓存的大小，单位为 4KB。如果 BUFFER 的值没有指定，则使用 DBM 中的参数 BACKBUFSZ 作为内部缓存的大小。进行备份时，数据首先由表空间写入内部缓存，当缓存满后，缓存中的数据会被写入到外部目标磁盘中。

BUFFER 值的大小建议设置为表空间 EXTENTSIZE 大小的整数倍。如果不同表空间的 EXTENTSIZE 大小不同，则 BUFFER 值建议为它们的最小公倍数。

（4）PARALLELISM n

PARALLELISM 参数指定了 db2bm（备份和恢复缓冲区操纵程序）进程数目，它决定了从表空间写入系统缓存的并行度。每一个 db2bm 进程会对应于一个或多个表空间，当一个 db2bm 进程完成对当前表空间的备份后，它会继续对另外一个表空间进行备份。由于每一个 db2bm 进程至少对应于一个表空间，因此，PARALLELISM 值需要小于数据库表空间的数目，如果将 PARALLELISM 设置为大于数据库中表空间的数目，并不会显著提高备份的性能。

比较：db2bm 进程和 db2med 进程

DB2 备份和恢复主要由 db2bm 进程和 db2med 进程共同完成，这两个进程均在 db2agent 的协调下进行工作。

- db2bm（备份和恢复缓冲区操纵程序）：用来在表空间和系统缓存间进行数据传输，它需要通过 db2pfchr（缓冲池预取进程）从容器中预取数据，或者通过 db2pclnr（缓冲池页清除程序）向容器中写入数据。
- db2med（备份和恢复介质控制器）：用来在系统缓存和外部介质之间进行数据传输。

（5）COMPRESS

DB2 使用的是一种改进的 Lempel-Zev（LZ）压缩算法，使用这个参数能减少备份文件的大小，从而节省磁盘空间，但是会使得 CPU 利用率提高。

7.4.2 DB2 崩溃恢复、版本恢复和前滚恢复与优化

数据库恢复是数据库备份的逆过程，也是数据库备份的目的，如果不需要恢复数据库，也就不需要备份数据库，针对不同的意外情况，DB2 提供了不同的恢复方法。与前面讲的数据库备份类似，DB2 同样集成了数据库恢复工具（Restore Utility），如下例所示，首先执行版本恢复，再执行前滚恢复：

```
--版本恢复
db2 restore db SAMPLE from /db2/db2lde taken at 20110407220438
--前滚恢复
db2 rollforward db SAMPLE to end of logs and complete
```

那么什么是版本恢复？什么是前滚恢复？从图 7-11 可以看到，DB2 提供三种恢复方式，即崩溃恢复（Crash Recovery）、版本恢复（Version Recovery）和前滚恢复（Rollforward Recovery）。

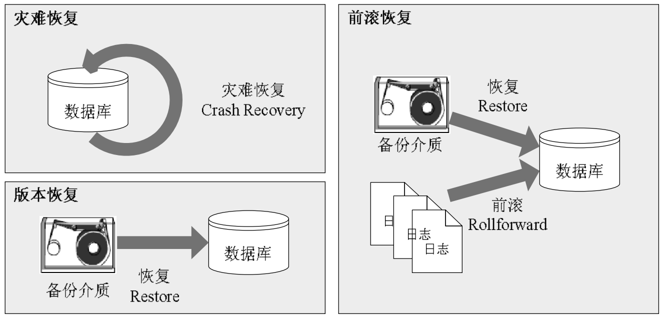


图 7-11　数据库恢复方式

1. 崩溃恢复

DB2 的灾难恢复是通过定时向数据库目录中的日志控制文件 SQLOGMIR.LFH、SQLOGCTL.LFH.1 和 SQLOGCTL.LFH.2 写入检查点，来控制崩溃恢复。在 DB2 重新启动后，会根据日志文件的内容将已经落实但未写到磁盘上的事务重新写到磁盘上去，而将所有未落实事务的变更回退到变更前的状态，通常由数据库自动进行。

2. 版本恢复

图 7-12 描述了如何进行版本恢复，即使用在某个时间点所备份的备份介质，把当前数据库恢复到之前那个时间点的数据库状态（我们可以把“版本恢复”理解为“时间点恢复”）。这也意味着从之前那个时间点，到当前时间点的数据库更新将被丢弃。我们可以对使用循环日志的数据库（即没有开启归档模式的数据库）使用此方法。当恢复完成后，数据库处于 Normal 状态，我们无须再进行额外的操作。

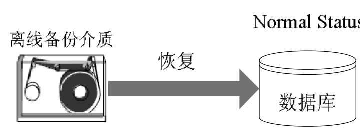


图 7-12　数据库版本恢复

3. 前滚恢复

首先使用备份介质将数据库恢复到之前的某个版本（相当于做了一次版本恢复），再根据归档日志将数据库恢复到某个特定的时间点，或者恢复到故障前的状态。如果使用前滚恢复，则数据库需要开启归档日志模式。

图 7-13 描述了如何进行前滚恢复：使用在线备份介质恢复完毕后，数据库处于 Rollforward Pending 状态，数据库等待前滚操作。

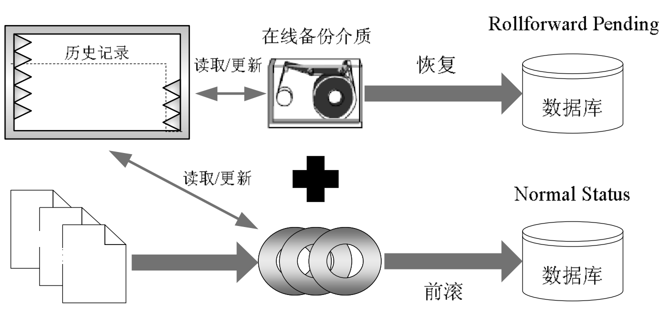


图 7-13　数据库前滚恢复

接下来，使用 Rollforward 命令前滚在此备份介质之后的归档日志，将数据库恢复到某个指

定的时间点或者灾难前的状态。

当前滚恢复完成后，我们需要使用 Rollforward Complete 命令重置数据库状态，将数据库由 Rollforward Pending 状态改为 Normal 状态。

在进行数据库恢复与前滚时，数据库恢复工具会读取备份恢复历史记录文件（db2rhist.asc），来决定使用哪些备份介质和哪些归档日志进行前滚恢复。

学习了恢复的基本原理后，下面重点讲述如何使用 RESTORE 和 ROLLFORWARD 命令进行版本恢复和前滚恢复。

使用备份介质进行版本恢复，命令如下所示。其中，SAMPLE 为我们需要恢复的数据库名字，关键字 from 指定了备份介质所在的目录，关键字 taken at 指定使用哪个备份介质。

```
[db2inst1@db2 ~]$db2 restore db SAMPLE from /db2/db2lde taken at 20111105012931
DB20000I  The RESTORE DATABASE command completed successfully.
```

通过以上命令就可以将数据库恢复到备份的状态了。那么数据库备备份之后发生的修改如何恢复呢？如果数据库处于归档模式，就可以通过下面的前滚恢复来实现了。

版本恢复结束后，如果尝试连接 SAMPLE 数据库会发现数据库返回警告信息，提示当前数据库处于 ROLLFORWARD PENDING 状态。

```
[db2inst1@db2 ~]$db2 connect to SAMPLE
SQL1117N  A connection to or activation of database "SAMPLE" cannot be made
because of ROLL-FORWARD PENDING.  SQLSTATE=57019
```

接下来使用前滚命令 ROLLFORWARD 将数据库恢复到损毁前的那一时刻。其中，SAMPLE 是当前需要进行前滚的数据库名字，关键字 to 指定了所需前滚的时刻，END OF LOGS 表示期望数据库前滚至最后一个日志文件结束，也就是将数据库前滚到损毁前的那一刻。

```
[db2inst1@db2 ~]$db2 rollforward db SAMPLE to end of logs

                                 Rollforward Status
  Input database alias                   = SAMPLE
  Number of nodes have returened status = 1

  Node number                             = 0
  Rollforward status                     = DB  working
  Next log file to be read               = S0000003.LOG
  Log files processed                    = S0000002.LOG - S0000003.LOG
  Last committeed transaction           = 2011-07-27-11.24.42.000000 UTC

DB20000I  The ROLLFORWARD command completed successfully.
```

当前滚结束后，需要使用 ROLLFORWARD COMPLETE 命令结束数据库的 ROLLFORWARD PENDING 状态，使数据库进入到 Normal 状态。如下所示。

```
[db2inst1@db2 ~]$db2 rollforward db SAMPLE complete

                                 Rollforward Status
  Input database alias                   = SAMPLE
  Number of nodes have returened status = 1

  Node number                             = 0
```

```
  Rollforward status                      = not pending
  Next log file to be read                =
  Log files processed                     = S0000002.LOG - S0000003.LOG
  Last committeed transaction            = 2011-07-27-11.24.42.000000 UTC

DB20000I  The ROLLFORWARD command completed successfully.
```

用户也许是由于数据库错误操作而进行的数据库恢复，这个时候如果前滚恢复到END OF LOGS就会重复错误的操作，在这种情况下可以将数据库前滚恢复到发生错误前的时间，如下所示：

```
db2 rollforward db SAMPLE to 2011-07-27-10.23.42.000000
```

上面讲述了如何使用 RESTORE 和 ROLLFORWARD 命令进行恢复和前滚。但是当使用增量备份或 Delta 备份（是两种增量备份方式，其详解见 7.4.1 小节）。在数据库恢复时，不得不需要执行多次 RESTORE 和 ROLLFORWARD 命令，这可不轻松啊！

例如从星期日到星期五，每天都做一次备份，备份的方式包括全备份、Delta 备份和增量备份。当数据库在星期六出现故障需要恢复时，如果使用 RESTORE 和 ROLLFORWARD 命令进行数据库恢复，恢复操作很繁琐：需要先使用三种不同的备份介质进行数据库恢复，再使用星期六的日志文件进行前滚，才能将数据库恢复到星期六故障前的状态。

为了解决这个问题，DB2 提供了 RECOVER 恢复命令，RECOVER 命令相当于 RESTORE 和 ROLLFORWARD 命令的组合，它会根据数据库备份恢复历史记录文件确定需要使用哪些备份介质和数据库日志进行恢复和前滚，这样仅需一条 RECOVER 命令就可以非常容易地完成数据库恢复和前滚了。

下面的例子讲解 RECOVER 命令的用法：

```
[db2inst1@db2 ~]$db2 recover db SAMPLE

                                 Rollforward Status
  Input database alias                    = SAMPLE
  Number of nodes have returened status = 1

  Node number                              = 0
  Rollforward status                      = not pending
  Next log file to be read                =
  Log files processed                     = S0000004.LOG - S0000004.LOG
  Last committeed transaction            = 2011-07-27-21.45.30.000000 Local

DB20000I  The RECOVER command completed successfully.
```

命令结束后，查看恢复之后数据，可以发现完全备份之后更改的数据也可以看到。

掌握了恢复基本操作，通过应用以下恢复时的最佳实践，能够优化恢复过程：

执行恢复操作时，DB2 将自动为缓冲池个数、缓冲池大小和并行性设置最佳值。此值根据实用程序堆内存的数量、可用处理器数和数据库配置而定。因此，根据系统中可用的存储量，应考虑通过增大 UTIL_HEAP_SZ 配置参数来分配更多内存。

虽然 DB2 恢复操作是自动调优的，但是根据实际情况调整以下参数可以提高恢复的性能：

```
RESTORE DATABASE <DB> FROM dir | dev WITH num-buf BUFFERS
BUFFER buff-size PARALLELISM n
```

（1）FROM dir | dev

如果指定多个备份存储路径（取决于备份时目标路径的数目），如 dir1、dir2、dir3 等，这样会有多个 db2med 进程并行地将数据从外部介质写入内部缓存，避免使 I/O 设备控制器带宽超负荷，能够缩短完成一次恢复操作所需的时间。

（2）WITH num-buff BUFFERS

指定使用的内部缓存大小，与备份参数相同，其最好设置为 db2med 进程数目的 2 倍以提高恢复速度。

（3）BUFFER buf-size

BUFFER buf-size 的大小为一块内部缓存的大小，其应该设置为表空间 EXTENTSIZE 大小的整数倍。如果没有设置的话，则使用 DBM 配置中的 RESTBUFSZ 参数作为内部缓存的大小。

（4）PARALLELISM n

与 DB2 备份命令相同，PARALLELISM 参数用来指定恢复过程中的 db2bm 进程的数目，通过增加 PARRALLELISM 值可以提高内部缓存和表空间之间数据交换的并行性。

如果一个表包含大量的长型字段和 LOB 数据，那么恢复它会占用非常多的时间。通过将长型字段和 LOB 数据存储在单独的表空间中，不选择恢复包含该长型字段和 LOB 数据的表空间，可以减少完成恢复操作所需的时间。

注意：包含长型字段或 LOB 字段表的表空间恢复

对于包含长型字段和 LOB 数据的表，在常规类型数据和长型字段或 LOB 数据存储在不同的表空间的情况下，如果只备份包含常规类型数据的表空间，而未备份包含相关的长型字段或 LOB 字段的表空间，那么不能对包含常规数据的表空间执行时间点前滚恢复。要执行时间点恢复，必须将一个表的所有表空间同时前滚至同一个时间点。

7.5 运维工具的调速策略

运维工具在生产系统上运行时，会对应用有一定的影响，为了在不影响应用的前提下更好地运行这些运维工具，DB2 提供了不同级别的调速策略，下面具体讲述。

1. 使用 UTIL_IMPACT_LIM 数据库管理器配置参数定义总体策略

例如，可以通过下面的 get dbm cfg 命令获取 UTIL_IMPACT_LIM 的当前设置，当前值为 20，这说明 DBA 期望运行所有运维工具的工作负载之和不会超过系统总负载的 20%。

```
[db2inst1@db2 ~]$db2 get dbm cfg | grep UTIL_
Workload impact by throttled utilities(UTIL_IMPACT_LIM) = 20
```

通常情况下 UTIL_IMPACT_LIM 这个参数设置为 0~100 之间的一个合理值，如果设置为 100，那么不启用调速，从而允许运维工具可以利用所有资源。如果发现运维工具运行的时间过长，那么可以考虑增大 UTIL_IMPACT_LIM 的值，甚至可以将 UTIL_IMPACT_LIM 设置为 100。

例如，使用下面的命令将 UTIL_IMPACT_LIM 参数从 20 改为 50，这样运行所有运维工具的工作负载之和不会超过系统总负载的 50%。

```
[db2inst1@db2 ~]$db2 update dbm cfg using UTIL_IMPACT_LIM 50
```

2. 在运维工具中指定 UTIL_IMPACT_PRIORITY 选项

除了 UTIL_IMPACT_LIM 参数外，有时候还需要为不同的运维工具指定不同的优先级，这需要在命令中设置 UTIL_IMPACT_PRIORITY 选项来完成。

UTIL_IMPACT_PRIORITY 选项的范围为 0~100，其值设置为正数，表示将受 UTIL_IMPACT_LIM 参数的限制，并且运行在调速模式下，其中 1 表示最低优先级，100 表示最高优先级。如果在命令中不指定 UTIL_IMPACT_PRIORITY 选项，那么工具不再受 UTIL_IMPACT_LIM 参数的限制，而将运行在非调速模式，从而可以使用系统的所有资源。

例如，下面的 BACKUP 命令通过 UTIL_IMPACT_PRIORITY 选项指定的优先级为 30：

```
BACKUP DATABASE PROD TO /backup UTIL_IMPACT_PRIORITY 30
```

例如，下面的 BACKUP 命令没有使用 UTIL_IMPACT_PRIORITY 选项，那么将运行在非调速模式下：

```
BACKUP DATABASE PROD TO /backup
```

3. UTIL_IMPACT_PRIORITY 配置参数定义特定运维工具的优先级

除了在运维工具中使用 UTIL_IMPACT_PRIORITY 选项外，还有一种更方便的方式，即在运行中动态调整多个工具的优先级，具体命令的语法如下所示：

```
SET UTIL_IMPACT_PRIORITY FOR <utility-id> TO <priority>
```

其中，utility-id 为运维工具在运行时的 ID，可以通过 list utilities 命令列出；priority 用来定义指定运维工具的优先级，范围为 0~100，具体含义如下：

（1）priority 的值从 0 改为正数，那么将从非调速方式进入调速方式，将受 UTIL_IMPACT_LIM 参数的限制，性能将会受到限制。

（2）priority 的值从正数改为 0，那么从调速方式进入非调速方式，将不再受 UTIL_IMPACT_LIM 参数的限制，可以使用系统的所有资源，具有最好的性能。

（3）priority 的值增大，优先级变高，反之，优先级变低，当然也会继续受 UTIL_IMPACT_LIM 参数的限制。

例如，下面使用 list utilities 命令列出了当前系统正在运行的运维工具。其中，ID 为 1 的是 RUNSTATS 操作，Priority 为 60；ID 为 2 的是 BACKUP 操作，Priority 为 30。可以看出，运行 RUNSTATS 操作的优先级要比 BACKUP 高。

```
[db2inst1@db2 ~]$db2 list utilities
ID = 1
Type = RUNSTATS
Database Name = PROD
Description = krrose.some_table
```

```
Start Time = 07/19/2013 17:30:09.521324
Priority = 60
ID = 2
Type = BACKUP
Database Name = PROD
Description = online db
Start Time = 07/19/2013 17:32:09.622395
State = Executing
Invocation Type = User
Throttling:
Priority = 30
```

为了加快它们的执行速度，将 RUNSTATS 操作退出调速模式（priority 改为 0），将 BACKUP 操作的优先级从 30 增加到 70，如下命令所示：

```
[db2inst1@db2 ~]$SET UTIL_IMPACT_PRIORITY FOR 1 TO 0
[db2inst1@db2 ~]$SET UTIL_IMPACT_PRIORITY FOR 2 TO 70
```

使用 list utilities 命令再次检查它们的运行情况，从下面的输出可以看出已经成功完成了更改，这样它们的性能将会显著提高：

```
[db2inst1@db2 ~]$db2 list utilities
ID = 1
Type = RUNSTATS
Database Name = PROD
Description = krrose.some_table
Start Time = 07/19/2013 17:30:09.521324
Priority = Unthrottled
ID = 2
Type = BACKUP
Database Name = PROD
Description = online db
Start Time = 07/19/2013 17:32:09.622395
State = Executing
Invocation Type = User
Throttling:
Priority = 70
```

注意：都有哪些运维工具或操作支持调速

目前支持调速策略的运维工具不多，仅有 2 种，分别是 RUNSTATS 和 BACKUP。另外，表空间重平衡操作和异步索引清除（AIC）操作也支持调速。

7.6 小结

本章首先从操作系统参数设置优化开始，讲解了配置参数优化，包括操作系统参数、DBM 参数、DB 参数和 DB2 注册变量的设置与优化。接下来讲解了数据库优化的常用工具，包括统

计信息收集、表和索引的重组以及包的重新绑定，通过以上操作，使得数据库执行引擎能够正确而高效地运行。然后介绍了 DB2 提供的各种数据移动工具及其应用场景和优化，以帮助读者能够在实际应用中灵活而高效地移动自己的数据。另外，还介绍了备份和恢复工具的使用与优化，这是数据库安全的重要保证。最后，讲解了运维工具的调速策略，这是业务能稳定运行的可靠保证。

第8章 锁和日志优化

提起并发，大家并不陌生，但是许多 DBA 并不清楚如何实现真正高效的数据库并发，甚至有人说：“我们已经配备了当前最强劲的硬件，怎么系统跑得比蜗牛还慢？”

本章将解答上述问题，主要内容包括：如何设置合理的隔离级才能使并发最大化；如何避免死锁和不必要的锁冲突；当出现未预期的锁问题时，如何快速处理；DB2 的锁和 Oracle 的锁到底有哪些区别。

数据库中最重要的就是数据，日志是实现数据安全的重要机制，本章还将深入探讨 DB2 日志原理以及最佳实践。

8.1　DB2 锁机制解析

锁，在生活中和每一个人息息相关，小到抽屉锁、保险柜锁，大到防盗门锁、车锁，在生活中发挥着无可替代的作用。

同样，锁在数据库中也发挥着无可替代的作用。为了让多个用户并发访问数据库对象时能同时保持数据一致性，DB2 采用了锁机制，当然 DB2 中的锁要比生活中的锁更复杂。

> 注意：锁的概念
>
> 锁实际上是 DB2 中的一个数据结构，锁的结构包括：
>
> - 对象（object）：包括行、表、数据块、表空间和索引。最重要的是行和表上的锁。
> - 大小（size）：指锁的粒度，其中行级别的锁粒度最小。在这里，可以把粒度理解为被锁定的数据范围。粒度越小，被锁定的数据范围就越小。
> - 时间长度（duration）：即锁从获取到释放所经历的时间。
> - 模式（mode）：规定了锁和其他锁的兼容性，即当某个对象已被某种锁锁定时，其他哪种类型的锁请求可以被满足。

8.1.1　没有锁会发生什么

锁是为了保证事务的一致性，而事务是由一个或多条 SQL 组成的工作单元。这些 SQL 被视为一个整体，如果都执行成功，执行的结果就在数据库中生效；如果任何一条 SQL 执行失败，则整个工作单元被回滚，也就是说，已经执行成功的 SQL 对数据做的改动也将被撤销。

实际上，如果没有锁机制的保护，事务一致性就无法得到保障，将引起丢失更新、脏读、不可重复读和幻影读等现象。为了便于读者更好的理解这四种现象，我们以银行系统中两个事务同时访问账户信息表 account 为例，看看没有锁的话，会出现哪些现象。

1. 丢失更新（Lost Update）

当多个事务同时对一行数据进行修改操作时，只有最后一个事务的修改被保留下来，其他事务的修改都会丢失。如图 8-1 所示，事务 1 和事务 2 同时更新账户信息表中 id 为 1004 的账号，当前值为 2000。当事务 1 用它的新值更新该行以后（balance=2000+500），事务 2 又更新了同一行（balance=2000+1000），事务 1 提交后，事务 2 提交。则事务 1 所执行的更新操作就会丢失，只留下事务 2 更新的结果，即最终 balance 的值被更新为 3000（即 2000+1000），而不是用户期望的 3500（即 2000+500+1000）。

2. 脏读（Dirty Read）

所谓“脏读”指的是一个事务会读到另一个事务未提交的数据。所谓“未提交数据”，是指被修改但是还未被提交的数据，它们有可能被回滚恢复到原值。如图 8-2 所示，事务 1 把 ID 为 1004 的 balance 值增加 500，即 balance=2000+500，但是还未提交； 此时事务 2 开始查询 ID 为 1004 的 balance 值，读到的值是 2500。但是，事务 1 决定回滚，撤销更新。这样 ID 为 1004

的 balance 实际值是 2000，而事务 2 读到的并非正确值。

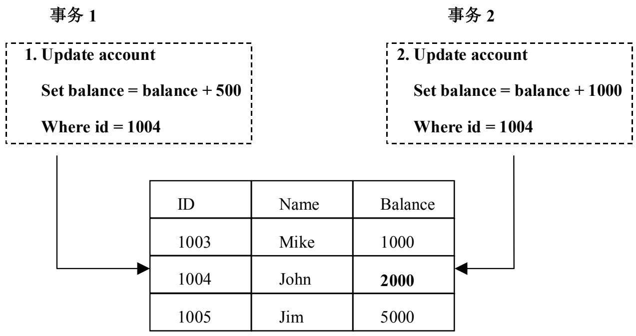


图 8-1　丢失更新示例

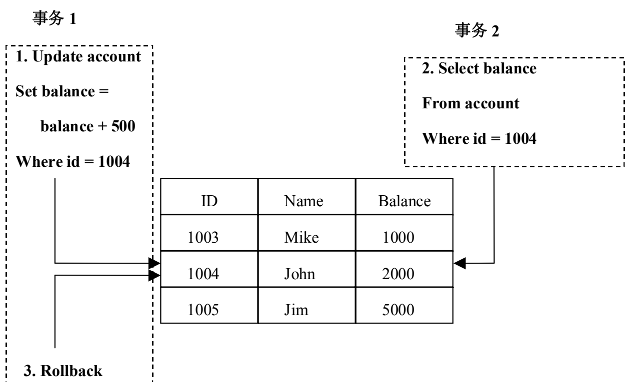


图 8-2　脏读示例

3. 不可重复读（Non Repeatable Read）

一个事务中两次读同一数据时，由于在两次读之间，其他事务修改了这个数据，造成该事务中两次读到的结果不同。如图 8-3 所示，事务 1 读取了一行数据（balance=2000），而此时事务 2 更改了该行（balance=2000+500），并立刻提交了更改。当事务 1 再次读取该行时，却读到了与上次不同的值（balance=2500)。对事务 1 来说，两次读到的信息产生了不一致现象。

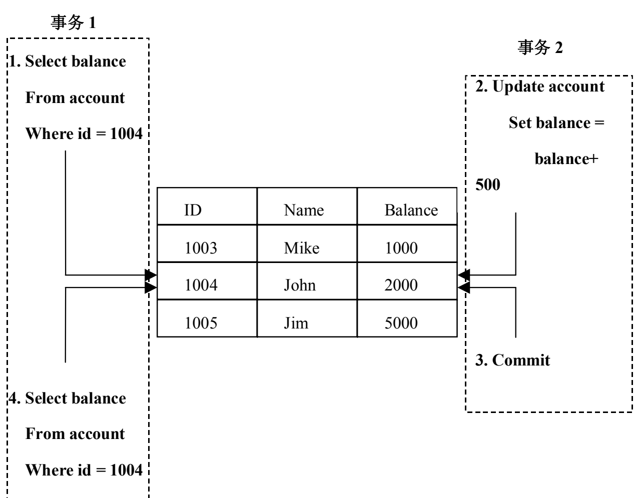


图 8-3　不可重复读示例

4. 幻影读（Phantom Read）

一个事务中两次读到的结果的行数不一样。第一次读操作后，由于其他事务做了插入、修改或删除数据行的操作，而使符合查询条件的数据行数发生了变化，该事务再次执行同样的读操作，就会发现结果集的大小增加了。如图 8-4 所示，事务 1 读取到满足查询条件 Balance>0 的 3 行数据后，事务 2 插入了符合此查询条件的新行并进行提交。此时事务 1 中的第二次查询，将会读到 4 行数据，即结果集增大了。

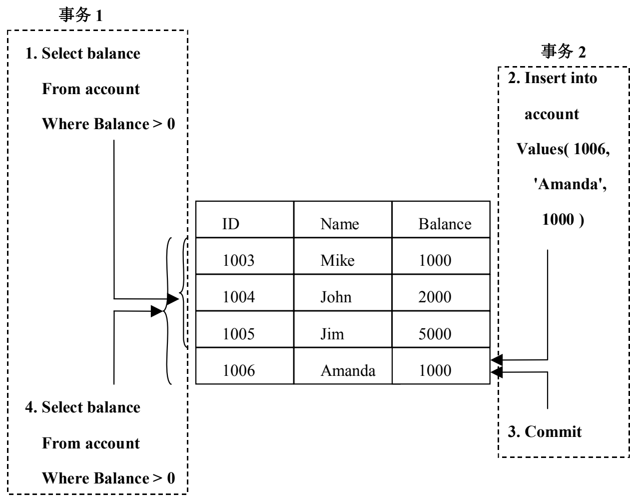


图 8-4　幻影读示例

比较：不可重复读和幻影读有什么区别？

不可重复读和幻影读都是执行同样查询却得到不同结果的现象。

在不可重复读现象中，第二次读的数据相较第一次发生了值的改变，但结果集的大小并没有改变。这种现象是由于其他事务修改（Update）了该事务第一次读到的数据行所致。

在幻影读现象中，用同样的查询条件第二次读取数据时，结果集大小发生了变化，这是由于修改了其他非结果集的数据或者插入新的数据行符合当前条件，导致结果集的数量变大所致。

根据业务逻辑的要求，用户一般不希望看到上述一种或多种现象出现，为此，DB2 使用了锁机制。有了锁机制后，应用程序对数据库对象进行操作时，DB2 会对相关的数据库对象加相应类型的锁。这样，操作同一对象的事务只有取得锁后才能进行相关操作，最终使执行结果满足事务一致性的要求。

8.1.2　锁的类型和兼容性

为了保证业务逻辑对事务一致性的要求，同时尽可能地保持最大并发能力，DB2 设计了很多类型的锁。每个刚学习 DB2 的人，都会觉得 DB2 锁的类型很多，如 S、IS、X、IX、SIX、NX、U、Z 等，名字上看区别不大，因此感觉比较复杂。其实，只要能都弄懂它们的含义就行了。

生活中锁有不同的粒度，比如小的保险柜锁和大的防盗门锁。DB2 中的锁也有不同的粒度，可以分为表锁和行锁，下面具体讲述。

1. 表锁

DB2 中有 8 种表锁，如表 8-1 所示。

表 8-1　表锁

表锁名	说明
IN	无意向锁（Intent None）
IS	意向共享锁（Intent Share）
IX	意向排他锁（Intent eXclusive）
SIX	带排他意向的共享锁（Share with Intent eXclusive）
S	共享锁（Share）
X	排他锁（eXclusive）
U	更新锁（Update）
Z	超级排他锁（superexclusive）

下面是每种表锁的具体含义。

- 无意向锁（IN）：事务一般是为了读取表中的数据（包括其他事务尚未提交的数据），而且无意对表进行修改而加此表锁。IN 锁一般用于“未提交读”（可读取未提交的数据）这一隔离级。
- 意向共享锁（IS）：事务为了读取表中部分数据行而加 IS 锁。锁需要与行锁配合，对应的行锁可以是 S 或 NS 锁。
- 意向排他锁（IX）：事务为了读取和修改表中的部分数据行而加此锁。IX 锁需要与行锁配合：对于读操作，对应的行锁可以是 S 锁、NS 锁、X 锁或 U 锁；对于更新操作，对应的行锁是 X 锁。
- 带排他意向的共享锁（SIX）：事务为了读取整个表，并修改其中的部分数据行而加此锁。SIX 锁对于更新操作，需要 X 行锁配合，而对于读操作不需要行锁配合。SIX 锁的获得比较特殊，它是在事务已经拥有 IX 锁的情况下，请求 S 锁；或者是已经拥有 S 锁的情况下，请求 IX 锁时生成的。

注意：为什么要有意向锁？

意向（intent）锁是对非意向锁的一种补充。意向锁要弱于对应的非意向锁。这里说的强弱，指的是对表锁的严格程度。例如，对一张表加锁，IX 是比 X 稍弱的锁，这是因为 IX 代表了表中的一部分行被锁住，而 X 表示整个表的所有行被锁住。IX 与 IX 是兼容的，因为它们可以代表对同一个表中的不同行进行了锁定；而 X 与 X 和 IX 锁都是不兼容的，因为 X 排斥其他 X 和 IX 锁对表的部分或全部锁定。

DB2 在对表加锁时，通过检查表上是否已经加了意向锁，就能知道表中是否有一部分行被其他事务锁住，而不需要逐行查看是否有事务对这张表加了锁，这自然节省了时间和系统资源。设想一下，如果一张表中有 1000 万行记录，那么意向锁的机制就大大缩短了执行时间。

可以这么说，意向锁是为了提高数据库并发性能而设计的。

- 共享锁（S）：事务为了读取整个表的数据而加 S 表锁。在表上加 S 锁后，不需要再在行一级加锁。如果一个事务得到某表的 S 锁，该事务可以读表中的任何数据；同时允许其他事务获得该表上的读锁。如果有事务需要更改该表上的数据，必须等待 S 锁被释放。
- 排他锁（X）：事务为了读取和更新整个表而加 X 表锁。其他事务不能更改表中数据，而且只有采用未提交读隔离级才可以读表中数据。如果在表上加了 X 锁，则不需要再在行一级加锁。
- 更新锁（U）：该锁是等待对表进行更改的一种中间状态。事务加 U 表锁后，可以读表中的任何数据，并最终可以通过获得表上的 X 锁来得到对表中任何数据的修改权。对于加了 U 锁的表，其他事务只能读取该表中的数据。值得一提的是，U 锁与 S 锁的区别主要在于更改的意图上。有了 U 锁，可以减小死锁发生的概率。例如，如果没有 U 锁，当两个事务在读取数据时，只能加 S 锁，然后这两个事务又都要对数据进行更新，这样两个事务都要将持有的 S 锁转换为 X 锁。为此，每个事务都需要等待对方先释放 S 锁，这就发生了死锁。有了 U 锁以后，上述两个事务，在一开始访问数据时就申请加 U 锁而不是 S 锁（如果是 Update 语句，DB2 会自动对数据加 U 锁；如果是 SELECT 语句，用户可以通过 FOR UPDATE 子句指定对读取的数据加 U 锁）。因为 U 锁之间是不相兼容的，其中一个事务获得 U 锁后，另一个事务将等待，这样就避免了死锁。
- 超级排他锁（Z）：事务一般在对表进行删除（Drop）、转换（Alter）操作，或是创建、删除索引的操作时而加 Z 锁。Z 锁不需要行锁配合。如果对表加上 Z 锁，则其他事务，包括采用未提交读隔离级的程序都不能对该表进行任何操作。

不同的表锁具有不同的锁定粒度，大家可以从它们之间的兼容关系体会其意义和用途。如表 8-2 所示，其中表锁 A 表示已加的一个表锁，表锁 B 表示待加的表锁，YES 或 NO 表示二者是否兼容。例如，IN 锁和 IN 锁对应 YES，表示二者兼容，即表上已经被加了 IN 锁的情况下，还可以加上另一个 IN 锁。

表 8-2　表锁的兼容关系

表锁 A	兼容性（表锁 B）							
	IN	IS	IX	SIX	S	U	X	Z
IN	YES	YES	YES	YES	YES	YES	YES	NO
IS	YES	YES	YES	YES	YES	YES	NO	NO
IX	YES	YES	YES	NO	NO	NO	NO	NO
SIX	YES	YES	NO	NO	NO	NO	NO	NO
S	YES	YES	NO	NO	YES	YES	NO	NO
U	YES	YES	NO	NO	YES	NO	NO	NO
X	YES	NO	NO	NO	NO	NO	NO	NO
Z	NO	NO	NO	NO	NO	NO	NO	NO

从表 8-2 还可以看出，Z 锁是最严格的，加 Z 锁意味着要对表的结构进行修改，例如，执行 Drop table 操作直接删除表。而 IN 锁是最不严格的表锁，与 Z 锁之外的其他锁都兼容，这意味着只要表结构不发生变化，就可以加锁成功。而 SIX 锁的兼容性等同于 S 锁加上 IX 锁。U

锁的作用是使有数据更改意向的事务互斥，同时保证读事务能读取数据。

2. 行锁

如果只在表级别上加上严格的表锁，那么锁的粒度就会过大，并发性将会很低。所以除表锁以外，DB2 还定义了行锁。行锁和表锁配合使用的方式，比单一使用表锁更加灵活，可以实现更佳的并发性能。DB2 中的行锁如表 8-3 所示。

表 8-3　行锁

行锁名	说明	要求的最低级别表锁
S	共享锁（Share）	IS
U	更新锁（Update）	IX
X	排他锁（eXclusive）	IX
NS	下一键共享锁（Next key Share）	IS
NW	下一键弱排他锁（Next key Weak exclusive）	IX

下面是对每种行锁具体含义的说明。

（1）共享锁（S）：事务为了读取数据行而加 S 行锁。它允许其他事务获得该行上的读锁（S、NS 或 U 锁）。如果其他事务需要更改该行，必须等 S 锁释放。该锁需要与表锁配合，需要的最低级别表锁是 IS。

（2）更新锁（U）：事务加 U 行锁是为了先读取数据行，然后通过升级到 X 锁，更改该数据行。和表上的 U 锁一样，该锁是等待对数据进行更改的一种中间状态。其他事务可以读数据，但不可以更新数据，行上的 U 锁主要是为了避免两个事务在拥有 S 行锁的情况下同时申请 X 行锁而造成死锁。该锁需要的最低级别表锁是 IX。

（3）排他锁（X）：事务为了修改数据行而加 X 行锁，其他事务不能对该行进行读或者更改操作。只有采用未提交读隔离级的事务才可以访问该行。该锁需要的最低级别表锁是 IX。

（4）下一键共享锁（NS）：事务为了读取数据行而加此锁。当 RS 和 CS 隔离级下（见 8.2.1 节对隔离级的详细介绍）读取数据行时，NS 锁会代替 S 锁（在 RR 隔离级下，仍加 S 锁）。该锁需要的最低级别的表锁是 IS。

（5）下一键弱排他锁（NW）：当一行数据被插入到建有索引的表中，该行在索引中对应的下一行会被加上该锁。该锁与 X 锁类似，只是与 NS 锁兼容。只有在表上有索引时才可能加 NW 锁。NW 行锁需要的最低级别的表锁是 IX。

上述行锁之间的兼容关系如表 8-4 所示，其中行锁 A 表示已加的一个行锁，行锁 B 表示待加的行锁，YES 或 NO 表示两者是否兼容。例如，S 锁和 S 锁对应 YES，表示两者兼容。

表 8-4　行锁的兼容关系

行锁 A	兼容性（行锁 B）				
	S	U	X	NS	NW
S	YES	YES	NO	YES	NO
U	YES	NO	NO	YES	NO

（续表）

行锁 A	兼容性（行锁 B）				
	S	U	X	NS	NW
X	NO	NO	NO	NO	NO
NS	YES	YES	NO	YES	YES
NW	NO	NO	NO	YES	NO

从表 8-4 可以看出，X 锁是最严格的，它屏蔽了所有其他锁。S 锁是最常用的锁，只和读操作的锁兼容。U 锁跟 S 锁相比，只是屏蔽了其他的 U 锁，用于有更改意图的操作之间的互斥，但不阻碍读操作。

注意：表级的意向锁是如何和行锁配合的？

如果表上已经有 S、U、X 或 Z 这样的表锁，这些表锁不需要和行锁配合。IS、IX、SIX 锁用于表级锁并且需要行锁配合，他们可以阻止其他事务对该表加排他锁。

- 如果一个事务获得某表的 IS 锁，则该事务可以获得表中某行的 S 锁，用于读取操作，同时其他事务也可以读取或更改表中的其他行。
- 如果一个事务获得某表的 IX 锁，则该事务可获得某一行上的 X 锁，用于更改操作，同时其他事务可以读取或更改表中的其他行。
- 如果一个事务获得某表的 SIX 锁，则该事务可以获得某一行上的 X 锁，用于更改操作，同时其他事务只能对表中其他行进行只读操作。

通过学习锁的类型、含义和互斥关系，就可以深入理解 DB2 的锁机制，下面讲述 DB2 的隔离级。

8.2　DB2 隔离级

当事务执行时，DB2 通过控制对数据对象加锁的类型和持有锁的时间，来最终实现隔离级指定的事务一致性要求。使用合理的隔离级可以确保数据库不会出现 8.1.1 节中列出一种或者多种不一致现象。

8.2.1　DB2 提供的四种隔离级

DB2 一共支持四种隔离级，按照严格程度，从低到高的顺序为：未提交读（Uncommitted Read）、游标稳定性（Cursor Stability）、读稳定性（Read Stability）和可重复读（Repeatable Read），隔离级越低，数据稳定性越差，但是并发性越好。

接下来，将结合 8.1.1 节中用到的例子，具体说明一下这些隔离级的含义。

1. 未提交读（UR）

最不严格的隔离级。在使用这个隔离级读取数据时，会对表加 IN 表锁，而不会对数据行

加锁。当然在事务试图修改数据时，仍然需要获取数据行上的 X 锁。

使用这种隔离级，不会发生丢失更新，但是会出现脏读、不可重复读和幻影读现象。

2. 游标稳定性（CS）

锁定游标定位的当前行（如果只是读取数据，则加 NS 锁；如果更改数据，则加 X 锁；如果是带有更改意向的读取，则加 U 锁）。对该行的锁定将持续到取出下一行记录或整个事务终止时为止。

这个隔离级解决了丢失更新和脏读的问题，但仍存在不可重复读和幻影读。它是 DB2 的默认隔离级，能够保证大多数业务场景下的一致性要求并提供最好的并发性。

> **小技巧：CS 隔离级在 DB2 V9.7 中的增强——当前已落实机制**
>
> 在 DB2 V9.7 中对 CS 隔离级进行了增强，即增加了当前已落实机制（即“Currently Committed”，也称为当前已提交）。
>
> V9.7 之前，在 CD 隔离级下，一个读事务在读某行数据时，如果该数据行被其他事务修改了，当修改它的事务还没有完成（提交或回滚），这个读事务就必须等待修改事务的完成。
>
> 如果启用了当前已落实，在这种情况下，这个读事务将不会等待修改事务的完成，直接读此行修改前的版本（即当前时刻处于已提交状态的版本）。这样大大增加了读事务和修改事务之间的并发性。
>
> 在 DB2 V9.7 中，“当前已落实”在缺省情况下是打开的。

3. 读稳定性（RS）

在使用这个隔离级时，只是锁定事务实际检索和修改的行（只是读取数据则加 NS 锁；修改数据加 X 锁；有修改意向的读取加 U 锁），其他不符合检索条件的行只是在被访问的时候被锁住，在访问下一行时会被释放。如果一个事务访问了 100 行，但符合检索条件的只有 10 行，那么只有这 10 行被锁定。

RS 隔离级可以解决丢失更新、脏读和不可重复读的问题，但仍存在幻影读的问题。

4. 可重复读（RR）

最严格的隔离级，可以防止所有数据不一致现象，即不出现脏读、不可重复读、幻影读和丢失更新现象，但是并发性能最差。

RR 隔离级保证在一个事务中，当两次执行同一个 SELECT 语句时，查询的结果总是相同的。为此，DB2 会对表上加 S 锁，直到事务结束时释放锁。

在 8.1.1 节的例子中，如果使用了 RR 隔离级，在操作账户信息表时，就不用担心其他事务插入新的账户记录或更新已读取的账户信息。

表 8-5 是隔离级和事务一致性现象的关系对应表。可以看到，丢失更新在任何一种隔离级中都不会出现；脏读会出现在未提交读隔离级中；不可重复读会出现在游标稳定性和未提交读隔离级中；幻影读只有在可重复读隔离级下才不会出现。

表 8-5　隔离级和事务一致性

隔离级	丢失更新	脏读	不可重复读	幻影读
可重复读（RR）	不出现	不出现	不出现	不出现
读稳定性（RS）	不出现	不出现	不出现	出现
游标稳定性（CS）	不出现	不出现	出现	出现
未提交读（UR）	不出现	出现	出现	出现

8.2.2　如何设定隔离级

DB2 提供了多种设定和调整隔离级的手段，隔离级可以在下列不同层次上设定：会话级（Session，也称为程序级）、连接级（Connection）和语句级（Statement）。

注意：在不同层次设定有什么不同？

根据所起作用的范围排序，会话级>连接级>语句级，但是越排在前面，优先级就越低。也就是说，会话级的设置会被连接级和语句级覆盖，连接级的设置会被语句级覆盖。

1. 会话级设置

（1）在 db2cli.ini 文件中使用 TXNISOLATION 关键字指定隔离级，可以改变 ODBC/JDBC 程序的隔离级。

（2）对于静态 SQL 语句，在执行 PREP（预编译命令）或 BIND （绑定命令）进行预编译或绑定时，可以使用 ISOLATION 选项指定隔离级。

（3）在 ODBC/CLI 中，使用 CHANGE ISOLATION LEVEL 命令改变会话隔离级。

2. 连接级设置

（1）在 ODBC/CLI 中，利用 SQLSetConnectAttr 函数，通过配置连接句柄的 SQL_ATTR_TXN_ISOLATION 属性，设置 CLI 连接的隔离级。该属性可以设置为如下的值：SQL_TXN_READ_UNCOMMITTED（未提交读）、SQL_TXN_READ_COMMITTED（相当于 DB2 的游标稳定性隔离级）、SQL_TXN_REPEATABLE_READ（相当于 DB2 的读稳定性隔离级）或 SQL_TXN_SERIALIZABLE（相当于 DB2 的可重复读隔离级）。例如，下面的 SQLSetConnectAttr 调用把连接句柄 hdbc 的 SQL_ATTR_TXN_ISOLATION 设定为 SQL_TXN_READ_ UNCOMMITTED，从而将 hdbc 对应的 CLI 连接的隔离级设置为未提交读：

```
cli_rc = SQLSetConnectAttr(hdbc,
                           SQL_ATTR_TXN_ISOLATION,
                           (SQLPOINTER)SQL_TXN_READ_UNCOMMITTED,
                           SQL_NTS);
```

（2）使用 SET CURRENT ISOLATION 语句对会话中发出的动态 SQL 语句设置隔离级。一旦设置 CURRENT ISOLATION 专用寄存器，就会影响会话中任何后续动态 SQL 语句的隔离级。此隔离级将一直有效直到会话结束或者发出 SET CURRENT ISOLATION RESET 语句为止。

3. 语句级设置

（1）在 ODBC/CLI 中，利用 SQLSetStmtAttr 函数，通过配置语句句柄的 SQL_ATTR_STMTTXN_ISOLATION 属性，设置语句的隔离级。

（2）可以通过 WITH [UR | CS | RS | RR]子句设定语句的隔离级。例如，如果你只想查看一下表 T1 的行数，且不要求结果很精确，你可以使用下面的语句。这个语句通过设定最宽松的 UR 隔离级，可实现最好的性能。

```
SELECT COUNT(*) FROM T1 WITH UR
```

8.3 实战案例：锁问题分析和解决

在本书第 1 章讲过懒惰系统，它往往和锁问题有关。一方面，锁问题会导致应用程序出错；另一方面，它延长了 SQL 语句处理时间，浪费了系统资源。

先说一个案例的具体场景：某银行新业务系统上线不久，业务部门就报告，在业务高峰时系统变得很慢，许多用户请求都无法及时响应。于是，运维人员对数据库进行了健康检查，从检查结果中没有发现明显的 CPU 或 I/O 瓶颈，但是发现了一些锁问题。

那么什么是锁问题？如何分析和解决这些锁问题？本节将深入讲述。

8.3.1 什么是锁问题

在 DB2 中，锁问题有 4 种，分别是锁等待、锁超时、锁升级和死锁。所有这些锁问题都会导致对资源不必要的锁定，延长事务的执行时间，从而产生性能下降。

1. 锁等待

当一个事务试图访问数据对象时，一般需要对数据对象申请加锁。在申请锁时，如果该数据库对象已经被其他事务锁定，并且所加的锁与正在申请的锁不兼容，就会发生锁等待。此时当前事务会被放入一个等待队列，等待占用该锁的事务提交或回滚来释放锁。

锁等待是一种正常的现象，一般情况下不是问题。长时间的锁等待或者过多的锁等待才可能导致性能问题。当出现锁等待时，有可能是隔离级过于严格，导致事务占用了不必要的锁。

2. 锁超时

当发生锁等待时，如果锁住对象的事务没有提交或回滚，那么请求锁的事务就会一直等待。为了防止请求锁的事务无限期等待，DB2 提供了 LOCKTIMEOUT 数据库参数。如果该参数设置为 0~32767 之间的值（单位为秒），则当锁等待的时间超过这个参数的设定值，就会返回 SQLCODE=－911，原因代码 68，并回滚事务，这就是锁超时。当 LOCKTIMEOUT 的值设置为－1（这也是缺省值）时，请求锁的事务会无限期等待。

锁超时一般是由以下原因造成的：

（1）LOCKTIMEOUT 的值太低；

（2）某个事务持有锁的时间过长。

3. 锁升级

DB2 中的锁本质上是内存里面的数据结构，随着数据库中对象上的锁不断增加，锁所占用的总内存也会不断增加。为了防止锁占用的内存无限制增加，在所占空间达到一定上限时，DB2 会试图把几个行级别的锁合并为一个表级别的锁从而释放锁空间，这就是锁升级。下面两个参数决定了锁所占空间的上限：

（1）LOCKLIST 参数

规定了锁列表所能够占用的总内存空间，默认值为 Automatic，即由 DB2 自动调节。注意，只有在 DB2 的数据库参数 SELF_TUNING_MEM 设为 ON，才会真正进行自动调节，对于下面的 MAXLOCKS 参数也是如此。

（2）MAXLOCKS 参数

规定了每个应用程序所能拥有的锁内存占锁列表的最大百分比，默认值为 Automatic，即由 DB2 自动调节。

当 DB2 锁列表耗尽或者一个应用程序所拥有的锁内存空间大于 LOCKLIST×（MAXLOCKS/100）个页的时候，就会发生锁升级。

锁升级一般来说是不正常的。虽然锁升级本身并不耗费多少时间，但是锁升级导致更多的应用程序阻塞，大大降低并发性，造成数据库性能下降，还会间接导致其他的锁问题，比如锁超时。所以，应该尽量避免锁升级。

4. 死锁

死锁就是两个或多个事务为获取对方拥有的资源而无限期相互等待的情况。出现无限期等待，是因为相互死锁的事务都正好持有对方想要的资源。比如，事务 A 获得了表 T1 上的锁，同时发出对表 T2 的锁请求，而事务 B 持有表 T2 上的锁，却同时发出对表 T1 的锁请求。这样，事务 A 要等待事务 B 提交并释放 T2 上的锁，同时事务 B 要等待事务 A 提交并释放 T1 上的锁，双方陷入了循环等待，从而产生死锁。

小技巧：通过一个春晚小品来理解死锁

大家还记得央视春晚中那脍炙人口的小品《开锁》吗？黄宏把所有的证件都存在了一个箱子里面，结果钥匙丢了，请来了开锁公司的人林永健来开锁。开锁公司的人需要客户出示有效证件才能打开箱子，黄宏需要打开箱子才能拿到和出示有效证件，双方都持有对方想要的资源且都不能释放，这样就形成了一个逻辑上的死锁。

当发生死锁时，事务可能会无限期地等待，也可能被死锁监视器检测到，并触发事务回滚。

8.3.2 从监控开始

当出现锁问题时，首先需要从监控开始。打开 bankdb 数据库的锁信息快照开关，收集快照信息后，发现系统存在严重的锁等待、锁超时、锁升级和死锁等问题：

```
=> db2 update monitor switches using lock on  ←打开锁快照开关

=> db2 get snapshot for database on bankdb     ←获取数据库 bankdb 上的快照
```

```
              Database Snapshot

    Database name                               = BANKDB
    ………
    Locks held currently                         = 15
    Lock waits                                   = 80          ←锁等待数
    Time database waited on locks (ms)          = 3350383     ←锁等待时间
    Lock list memory in use (Bytes)              = 54400
    Deadlocks detected                           = 11          ←检测到的死锁数
    Lock escalations                             = 20          ←锁升级数
    Exclusive lock escalations                   = 0
    Agents currently waiting on locks            = 10
    Lock Timeouts                                = 134         ←锁超时数
    Number of indoubt transactions               = 0
```

通过检查系统诊断日志 db2diag.log，进一步确认了锁超时、锁升级等出错信息：

```
    2011-07-01-13.33.27.263057+480 I38346815E819      LEVEL: Info
    PID     : 3735                TID  : 47467592673600PROC : db2sysc 0
    INSTANCE: inst1               NODE : 000          DB   : BANKDB
    APPHDL  : 0-52                 APPID: *N0.user16.110729053249
    AUTHID  : USER16
    EDUID   : 37                  EDUNAME: db2agent (BANKDB) 0
    FUNCTION: DB2 UDB, oper system services, sqlofica, probe:10
    DATA #1 : SQLCA, PD_DB2_TYPE_SQLCA, 136 bytes
     sqlcaid : SQLCA     sqlcabc: 136   sqlcode: -911   sqlerrml: 2
     sqlerrmc: 68
     sqlerrp : SQLRI2C4
     sqlerrd : (1) 0x80100044      (2) 0x00000044      (3) 0x00000000
              (4) 0x00000000      (5) 0xFFFFFE0C      (6) 0x00000000
     sqlwarn : (1)      (2)      (3)      (4)        (5)       (6)
              (7)      (8)      (9)      (10)       (11)
     sqlstate: 40001

    。。。。
    2011-07-07-14.18.13.834104 Instance:user16 Node:000 DB: BANKDB
    PID:81912(db2agent ) TID:1 Appid: *N0. user16.110729053249
    ADM5502W The escalation of "635" locks on table "ACCOUNTS_BALANCE" to lock intent "S"
was successful

    。。。。
    2011-07-09-10.58.28.094844-300 I37273015A836      LEVEL: Error
    PID     : 594116              TID  : 9661        PROC : db2sysc 0
    INSTANCE: inst1               NODE : 000         DB   : BANKDB
    APPHDL  : 0-878                APPID: *N0.user14.110802155747
    AUTHID  : USER14
    EDUID   : 9661                EDUNAME: db2agent (PROD) 0
    FUNCTION: DB2 UDB, trace services, sqlt_logerr_data (secondary logging func
    DATA #1 : SQLCA, PD_DB2_TYPE_SQLCA, 136 bytes
     sqlcaid : SQLCA     sqlcabc: 136   sqlcode: -911   sqlerrml: 1
     sqlerrmc: 2
     sqlerrp : SQLRI2C4
     sqlerrd : (1) 0x80100002      (2) 0x00000002      (3) 0x00000000
```

```
        (4) 0x00000000     (5) 0xFFFFFE0C     (6) 0x00000001
sqlwarn : (1)      (2)      (3)      (4)       (5)      (6)
        (7)      (8)      (9)      (10)       (11)
sqlstate:….
```

应用程序也不时返回－911 错误：

```
SQL0911N  The current transaction has been rolled back because of a deadlock
or timeout.  Reason code "68".  SQLSTATE=40001
```

读者可能会问：这些锁等待、锁超时和锁升级是否合理？隔离级是否设置正确？是应用程序设计的问题还是数据库相关参数配置的问题？下面将深入探讨分析与解决这些问题。

8.3.3　锁等待分析和解决

在上一节的案例中，发现系统有多次锁等待。通过下面的步骤，可以按部就班的对锁等待进行分析。

（1）通过 db2pd 命令查看数据库 bankdb 中锁等待的详细情况。

```
=> db2pd -db bankdb -wlocks
Database Member 0 -- Database BANKDB -- Active -- Up 0 days 00:10:20 -- Date 06/09/2011
Locks being waited on :
AppHandl TranHdl   Type      Mode  Sts  AppName  AppID
3511       14     RowLock    ..X    G    db2bp   *LOCAL.user.110609204956
3509       15     RowLock    ..U    W    db2bp    *LOCAL.user.110609204832
```

注意上面的输出中的 Sts 列，它表示锁的状态，“G”表示 granted，即已获得的锁；“W”表示正在等待的锁。由 Sts 列可以看出，一个 TranHdl 为 14 的事务（对应的应用程序句柄为 3511）获得了该对象的锁（X 锁），另一个 TranHdl 为 15 的事务（对应的应用程序句柄为 3509）则在等待对该对象加锁（U 锁）。

（2）通过 db2pd–apinfo 命令可以看到两个事务所对应的应用程序（句柄分别为 3511 和 3509）的详细信息：

应用程序 3511 的详细信息，可以看出事务 14 处于等待完成状态：

```
=> db2pd -apinfo 3511 -db bankdb
Database Member 0 -- Database BANKDB -- Active -- Up 0 days 00:37:38
apinfo Time:    07/09/2011 16:18:30
Application :
  Address :              0x07800000048B0080
  AppHandl [nod-index] :   3511     [000-03511] ←应用程序句柄
  TranHdl :              14
  Application PID :        885140
  Application Node Name :  maradona
  IP Address:             n/a
  Connection Start Time :  (1307652596)Thu Jun  9 15:49:56 2011
  Client User ID :         user
  System Auth ID :         USER
  Coordinator EDU ID :     8537
  Coordinator Member :  0
  Number of Agents :      1
```

```
    Locks timeout value :    NotSet
    Locks Escalation :       No
    Workload ID :           1
    Workload Occurrence ID : 4
    Trusted Context :       n/a
    Connection Trust Type :  non trusted
    Role Inherited :        n/a
    Application Status :     UOW-Waiting  ←该应用程序处于 UOW-Waiting 即等待事务完成状态
    Application Name :       db2bp
    Application ID :        *LOCAL.user.110609204956
    ......
```

从下面应用 3509 的详细信息，可以看出事务 15 正在执行将账户（ACCOUNT_ID 为"95533222"）的余额增加 3000 元的操作，而应用程序的状态处于 Lock-wait 也就是锁等待状态。从步骤（1）中已经看到，这个锁等待是由于事务 14 以 X 锁锁定该账户，导致应用程序 3509 中的事务 A 无法获得 U 锁造成的。

```
  => db2pd -apinfo 3509 -db bankdb

  Database Member 0 -- Database BANKDB -- Active -- Up 0 days 00:48:40

  apinfo Time:   07/09/2011 16:29:32

  Application :
    Address :              0x07800000048D0080
    AppHandl [nod-index] :   3509    [000-03509] ←应用程序句柄
    TranHdl :              15
    Application PID :        610424
    Application Node Name :  maradona
    IP Address:             n/a
    Connection Start Time :  (1307652512)Thu Jun  9 15:48:32 2011
    Client User ID :        user
    System Auth ID :        user
    Coordinator EDU ID :     9306
    Coordinator Member :  0
    Number of Agents :       1
    Locks timeout value :    NotSet
    Locks Escalation :       No
    Workload ID :           1
    Workload Occurrence ID : 5
    Trusted Context :       n/a
    Connection Trust Type :  non trusted
    Role Inherited :        n/a
    Application Status :     Lock-wait          ←该应用程序处于 Lock-wait 即锁等待状态
    Application Name :       db2bp
    Application ID :        *LOCAL.user.110609204832
   ......
    List of active statements :
     *UOW-ID :         1
      Activity ID :     1
      Package Schema :  NULLID
      Package Name :    SQLC2J22
      Package Version :
```

```
    Section Number :  203
    SQL Type :        Dynamic
    Isolation :       CS
    Statement Type :  DML, Insert/Update/Delete
    Statement :       update ACCOUNTS_BALANCE SET   ←该应用程序正在执行的 SQL 语句
    BALANCE = BALANCE + 3000 WHERE ACCOUNT_ID ='95533222'
```

（3）通过 db2pd –dynamic 查看 bankdb 上执行过的语句，从而确认事务 14 执行过什么语句。从下面的输出中，可以发现两个语句：一个是前面看到事务 15 正在执行的 Update 语句；另一个就是要找的事务 14 执行过的 Update 语句，正是这个语句修改了账户，锁住了对应数据行。

```
=>  db2pd -db bankdb -dynamic
Database Member 0 -- Database BANKDB -- Active -- Up 0 days 01:02:51
Dynamic Cache:
Current Memory Used           1432076
Total Heap Size              13699317
Cache Overflow Flag           0
Number of References          1547
Number of Statement Inserts   507
Number of Statement Deletes   473
Number of Variation Inserts   86
Number of Statements          34

Dynamic SQL Statements:
Address        AnchID StmtUID    NumEnv     NumVar     NumRef     NumExe     Text
  0x070000003757FD00 6      1          1         1           1         1

  update ACCOUNTS_BALANCE SET  BALANCE = BALANCE + 900 WHERE ACCOUNT_ID ='95533222'
  ←这是事务 14 已经执行过的 Update 语句

  0x070000003346DD00 7      1          1         1           1         1

  update ACCOUNTS_BALANCE SET  BALANCE = BALANCE + 3000 WHERE ACCOUNT_ID ='95533222'
  ←这是事务 15 正在执行的 Update 语句
......
```

（4）最后可以确定，两个事务同时转账到相同账户。事务 14 先将 900 元打入 ID 为 95533222 的账户，数据库会执行一个 Update 语句，更新此账户的余额。正如上面通过 db2pd-dyn 命令看到的，执行的 Update 语句如下：

```
update accounts_balance set balance = balance + 900 where account_id = '95533222'
```

这个时候，事务 14 还没有提交，事务 15 要向同一账户打入 3000 元。它执行了下列语句：

```
update accounts_balance set balance = balance + 3000 where account_id = '95533222'
```

注意：事务 14 为什么长时间持有锁？

事务 14 更新账户余额时，在 CS 隔离级下，数据库对被修改的账户加了 X 锁，而且这个锁会保持到事务结束。事务 15 在读到该账户时，要加 U 锁，但因为 U 锁和 X 锁不兼容，所以要等待事务 14 完成并释放锁。但是事务 14 由于应用程序的问题，需要较长时间才能执行完毕，这个时间段它所持有的锁不会释放，这样事务 15 只能处于锁等待状态。

从上面的分析中可以看到，在 CS 隔离级下，当一个事务在修改某一条记录，另一个事务试图修改同一条记录时会被阻塞。那么，另一个事务读取该记录是否会被阻塞呢？

由于 DB2 在 V9.7 中引入了“当前已落实”功能，只读的操作遇到被其他事务修改而未提交的记录时，会读取该记录修改前的版本，所以不会被阻塞。

8.3.4 锁超时分析和解决

在这个案例中，根据快照信息发现数据库发生了多次锁超时，相关信息如下：

```
=> db2 get snapshot for database on bankdb
            Database Snapshot
Database name                               = BANKDB
Lock Timeouts                                 = 134            ←锁超时数
```

查看数据库 bankdb 的 LOCKTIMEOUT 参数，发现设置为 30 秒。

```
=> db2 get db cfg |grep LOCKTIMEOUT
 Lock timeout (sec)                        (LOCKTIMEOUT) = 30
```

那么如何监控和分析这种锁超时事件呢？推荐使用锁事件监视器来监控锁超时事件，步骤如下：

（1）使用 CREATE EVENT MONITOR FOR LOCKING 语句创建名为 mon_timeout 的锁事件监视器：

```
CREATE EVENT MONITOR mon_timeout FOR LOCKING WRITE TO UNFORMATTED EVENT TABLE
```

（2）为了让 DB2 收集详细的锁信息，需要更改 MON_LOCKTIMEOUT、MON_ DEADLOCK、MON_LOCKWAIT 和 MON_LW_THRESH 数据库参数。

将前三个参数都设为 HIST_AND_VALUES，这样能够收集尽量多的信息。同时，MON_LW_THRESH 被设为 10 秒（下面命令中的数字 10000000 的单位是微秒）。

```
db2 update db cfg using MON_LOCKTIMEOUT HIST_AND_VALUES MON_DEADLOCK HIST_AND_VALUES
MON_LOCKWAIT HIST_AND_VALUES  MON_LW_THRESH 10000000
```

（3）激活锁事件监视器 mon_timeout。锁事件监视器创建成功后处于关闭状态，可以用下面的语句将监视器的状态设置为 1 来开启监视器。

```
set event monitor mon_timeout state 1
```

（4）当发生锁超时事件时，相关信息会被锁事件监视器写入 mon_timeout 表中。可以调用存储过程 EVMON_FORMAT_UE_TO_TABLES 将锁超时信息从表 mon_timeout 中取出，进行格式化后存入创建的新表中：

```
call EVMON_FORMAT_UE_TO_TABLES ('LOCKING', NULL, NULL, NULL, NULL, NULL,
'RECREATE_FORCE', -1, 'SELECT * FROM mon_timeout ORDER BY event_timestamp')
```

调用成功后，会在当前模式下创建下列 5 张新表：LOCKWAITEVM、LOCK_ ACTIVITY_ VALUES、LOCK_EVENT、LOCK_PARTICIPANTS 和 LOCK_PARTICIPANT_ ACTIVITIES，格式化后的锁超时信息就存放在这些表中。

（5）通过下面的语句可以得到锁超时对应的应用程序信息：

```
#查询发生的锁事件的 XML_ID 名，其中有锁事件的类型和时间等信息
select substr(XMLID,1,64) xml_id from LOCK_EVENT

 XML_ID
 ----------------------------------------------------------------
 db2LockEvent_2_LOCKWAIT_2011-05-18-17.34.09.293236_0
 db2LockEvent_3_LOCKTIMEOUT_2011-05-18-17.34.34.243606_0

#查询参与锁超时的应用程序
 select PARTICIPANT_NO as p_no, PARTICIPANT_TYPE as p_type,
 PARTICIPANT_NO_HOLDING_LK as p_no_holdling_lk, APPLICATION_HANDLE as
 appl_hanle, substr(TABLE_NAME,1,10) as t_name from LOCK_PARTICIPANTS

P_NO     P_TYPE    P_NO_HOLDLING_LK APPL_HANLE         T_name
----------------------------------------------------------------
 1       Requester    2                3511              ACCOUNTS_BALANCE
 2       Owner        -                3509              -
```

可以看到，句柄为 3511 的应用程序在请求句柄为 3509 的应用程序的锁，通过下面的 db2pd 命令可以分别查看这两个应用程序的情况：

```
db2pd -apinfo 3511 -db bankdb
db2pd -apinfo 3509 -db bankdb
```

（6）查看和锁相关的语句：

```
    SELECT   PARTICIPANT_NO,   ACTIVITY_TYPE,   substr(STMT_TEXT,1,256)   sql_text   FROM
LOCK_PARTICIPANT_ACTIVITIES
```

得到和锁相关的语句后，导致锁超时的原因就找到了：两个应用程序在各自的事务中更新了同一个账户，而且第一个应用程序的事务用下面的语句先对 ID 为“95533222”的账户进行了更新操作，在事务提交前，一直持有账户上的 X 锁。

```
update ACCOUNTS_BALANCE SET BALANCE = BALANCE + 3000 WHERE ACCOUNT_ID ='95533222'
```

然后第二个应用程序的事务进行同样的 Update 更新操作。第二个事务因为请求加 U 锁而要等待第一个事务的 X 锁释放，进而陷入锁等待。

由于锁超时 LOCKTIMEOUT 设置为 30 秒，所以 30 秒后，触发了锁超时，事务被回滚，得到 SQL0911N 错误消息。

至此，完成了对锁超时事件的分析。对于这个锁超时，可以通过尽快提交持有锁的事务，或者增加锁超时参数 LOCKTIMEOUT 来解决。

8.3.5　锁升级分析和解决

DB2 有一个很特别的锁问题，即锁升级，可以从快照信息中发现：

```
=> db2 get snapshot for database on bankdb
           Database Snapshot
```

```
Database name                          = BANKDB
Lock escalations                        = 20            ←锁升级数
```

那么该如何分析锁升级呢？

首先，通过获取数据库或者应用程序的快照，可以得到锁升级发生的次数。

其次，通过 DB2 诊断日志（db2diag.log）能够确定在哪些表上发生了锁升级。在这个案例中，db2diag.log 文件里面就出现了下列警告信息。可以看到，表 USER.ACCOUNT 中的 635 个行锁被升级成了表上的 S 锁。

```
2010-06-07-14.18.13.827913 Instance:user Node:000
PID:81912(db2agent (PROD)) TID:1 Appid:*LOCAL.user.050207191812
data management sqldEscalateLocks Probe:2 Database:BANKDB
ADM5500W DB2 is performing lock escalation. The total number of locks
currently held is "864", and the target number of locks to hold is "432".

2010-06-07-14.18.13.834104 Instance:user Node:000
PID:81912(db2agent (PROD)) TID:1 Appid:*LOCAL.user.050207191812
data management sqldEscalateLocks Probe:3 Database:BANKDB
ADM5502W The escalation of "635" locks on table "USER.ACCOUNT" to lock
intent "S" was successful
```

最后，通过增大 LOCKLIST 和 MAXLOCKS 的值以减少或者避免锁升级。还可以将它们都设为 AUTOMATIC，即由 DB2 自动根据需要管理锁空间。另外，从应用程序的角度来看，通过增加提交次数或者降低隔离级，也能够减少锁升级发生的次数。

8.3.6 死锁分析和解决

在这个案例中，从快照信息中可以看出，发生了 11 次死锁，相关信息如下：

```
=> db2 get snapshot for database on bankdb
            Database Snapshot
Database name                          = BANKDB
Deadlocks detected                      = 11            ←检测到的死锁数
```

发现了死锁，该如何解决呢？

第一种办法，如果死锁可以用固定的步骤重现，那么可以通过重现死锁，并运用分析锁等待的方法，确定哪些事务导致了死锁，最后通过调整事务的隔离级或者修改相应的 SQL 语句执行顺序来解决。

另外一种办法是，使用死锁事件监视器，在死锁发生时收集并保存相关的信息，下面使用这种方法分析一下这个案例的死锁问题。为叙述方便，我们对案例中实际的事务语句、表名以及表结构等做了简化。

（1）创建并打开死锁监视器 deadlockmon。

创建和打开死锁事件监视器的命令如下，其中，deadlockmon 是事件监视器的名字；write to file 选项指定了存放收集的信息的文件路径，即/home/user/event；autostart 选项指定此监视器在数据库启动时自动打开。

```
--创建死锁监视器 deadlockmon
create event monitor deadlockmon for deadlocks with details write to file
'/home/user/event' autostart

--将死锁监视器激活
set event monitor deadlockmon state 1
```

（2）创建死锁监视器后，DB2 每隔一定时间（由数据库参数 DLCHKTIME 定义）会自动检测系统中是否有死锁事件发生。

当发现死锁时，DB2 会选取参与死锁事件的某个事务进行回滚，直到死锁解除。在死锁被解除的同时，死锁相关的信息也被保存下来了。被回滚的事务将收到下面的出错信息：

```
DB21034E  The command was processed as an SQL statement because it was not a
valid Command Line Processor command.  During SQL processing it returned:
SQL0911N  The current transaction has been rolled back because of a deadlock
or timeout.  Reason code "2".  SQLSTATE=40001
```

（3）发生死锁后，使用下列命令得到死锁监视器收集的信息。

通过 flush event monitor 命令将收集的信息写回磁盘，随后使用 db2evmon 命令格式化收集到的信息，并写入文件 mydldata.txt 中。

```
--将 deadlockmon 收集的信息写回磁盘
flush event monitor deadlockmon

--关闭 deadlockmon
set event monitor deadlockmon state 0

--输出并格式化监控信息到文件 mydldata.txt 中（在/home/user/event 下执行）
db2evmon -db bankdb -evm deadlockmon >mydldata.txt
```

（4）打开 mydldata.txt，可以看到利用死锁监视器收集的信息，从中可以看出参与死锁的所有事务。死锁信息中出现的事务执行了语句“update table2 set b = 100 where a = 2”，通过对比应用程序名、用户信息等，我们发现它来自于运行在 bankdb 上的一个应用程序，在这里称之为应用程序 A：

```
Deadlocked Connection ...
  Deadlock ID:   3
  Participant no.: 2
  Participant no. holding the lock: 1
  Appl Id: *LOCAL.user.110703200820
  Appl Seq number: 00006
  Tpmon Client Applname: CLP test_join2
  Appl Id of connection holding the lock: *LOCAL.user.110703200705
  Seq. no. of connection holding the lock: 00001
  Lock wait start time: 07/03/2011 16:25:37.722982
  Lock Name       : 0x000000000000000000000000000
  Lock Attributes : 0x00000000
  Release Flags   : 0x00000000
  Lock Count      : 0
  Hold Count      : 0
  Deadlock detection time: 07/03/2011 16:25:52.523712
```

```
    Table of lock waited on      : TABLE2
    Schema of lock waited on     : USER
    Data partition id for table  : 0
    Tablespace of lock waited on : USERSPACE1
    Type of lock: Row
    Mode of lock: X   - Exclusive
    Mode application requested on lock: X   - Exclusive
    Node lock occured on: 0
    Lock object name: 5
    Application Handle: 670
    Deadlocked Statement:
      Type     : Dynamic
      Operation: Execute Immediate
      Section  : 203
      Creator  : NULLID
      Package  : SQLC2J22
      Cursor   :
      Cursor was blocking: FALSE
      Text     : update table2 set b = 100 where a = 2
    List of Locks:
        Lock Name                   : 0x000000020000000100011FA0D6
        Lock Attributes             : 0x00000000
        Release Flags               : 0x40000000
        Lock Count                  : 1
        Hold Count                  : 0
        Lock Object Name            : 4
        Object Type                 : Row
        Tablespace Name             : USERSPACE1
        Table Schema                : USER
        Table Name                  : TABLE1
        Data partition id           : 0
        Mode                        : X   - Exclusive
```

下面的死锁事务执行了语句“update table1 set b = 200 where a = 1”，这里将执行这个事务的应用程序称为应用程序 B：

```
  Deadlocked Connection ...
    Deadlock ID:   3
    Participant no.: 1
    Participant no. holding the lock: 2
    Appl Id: *LOCAL.user.110703200705
    Appl Seq number: 00003
    Appl Id of connection holding the lock: *LOCAL.user.110703200820
    Seq. no. of connection holding the lock: 00001
    Lock wait start time: 07/03/2011 16:25:46.247390
    Lock Name       : 0x000000000000000000000000000
    Lock Attributes : 0x00000000
    Release Flags   : 0x00000000
    Lock Count      : 0
    Hold Count      : 0
    Deadlock detection time: 07/03/2011 16:25:52.525562
    Table of lock waited on      : TABLE1
    Schema of lock waited on     : USER
```

```
    Data partition id for table  : 0
    Tablespace of lock waited on : USERSPACE1
    Type of lock: Row
    Mode of lock: X   - Exclusive
    Mode application requested on lock: X   - Exclusive
    Node lock occured on: 0
    Lock object name: 4
    Application Handle: 658
    Deadlocked Statement:
      Type     : Dynamic
      Operation: Execute Immediate
      Section  : 203
      Creator  : NULLID
      Package  : SQLC2J22
      Cursor   :
      Cursor was blocking: FALSE
      Text     : Update table1 set b = 200 where a = 1
    List of Locks:
        Lock Name                   : 0x00000002000000100010BA0D6
        Lock Attributes              : 0x00000000
        Release Flags               : 0x40000000
        Lock Count                  : 1
        Hold Count                  : 0
        Lock Object Name             : 0
        Object Type                 : Internal - Variation
        Data partition id            : -1
        Mode                       : S   - Share

        Lock Name                   : 0x00020104000000000000000552
        Lock Attributes              : 0x00000000
        Release Flags               : 0x40000000
        Lock Count                  : 1
        Hold Count                  : 0
        Lock Object Name             : 5
        Object Type                 : Row
        Tablespace Name              : USERSPACE1
        Table Schema                : USER
        Table Name                  : TABLE2
        Data partition id            : 0
        Mode                       : X   - Exclusive
```

仔细查看应用程序 A 和 B 中的语句，发现它们首先分别更新了表 table1 和表 table2，并持有更新行的 X 锁:

```
--应用程序 A
update table1 set b = 5 where a = 1

--应用程序 B
update table2 set b = 4 where a = 2
```

然后应用程序 A 更新表 table2，应用程序 B 更新表 table1：

```
--应用程序 A
update table2 set b = 100 where a = 2
```

```
--应用程序 B
update table1 set b = 200 where a = 1
```

如图 8-5 所示，应用程序 A 和 B 要更新两张表里面相同的行。应用程序 A 先更新了表 Table1 的行 Row1，然后试着更新表 Table2 的行 Row2。而应用程序 B 采用相反的顺序，先更新了 Table2 的 Row2，继而试着更新 Table1 的 Row1。

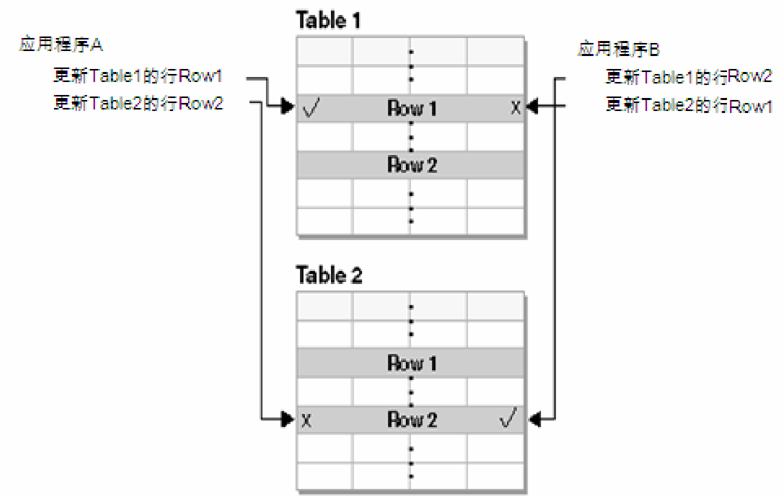


图 8-5　案例中的死锁情形

在 CS 隔离级下，此时应用程序 A 持有 Table1 中 Row1 上的 X 锁，而试着申请 Table2 中 Row2 上的 U 锁；应用程序 B 持有 Table2 中 Row2 上的 X 锁，而试着请求 Table1 中 Row1 上的 U 锁。

由于 U 锁和 X 锁不兼容，A 将处于等待 B 释放 Row2 上的 X 锁的状态，同时 B 将处于等待 A 释放 Row1 上的 X 锁的状态。这样，两个程序陷入死锁状态。

这个死锁实际上是由于两个事务执行操作的顺序有问题造成的。通过改变应用程序 A 或 B 中两个更新操作的顺序，让它们都按照先更新表 Table1 再更新表 Table2（或者相反）的顺序进行，就可以避免死锁。

8.4　深入浅出——DB2 日志的秘密

众所周知，为保证数据的正确性，数据库的事务都应该满足 ACID 原则：即原子性（Atomic）、一致性（Consistency）、完整性（Integrity）和持久性（Duration）。无论是 Oracle 数据库，还是 DB2 数据库，数据库日志机制是保证 ACID 原则的核心。

8.4.1　DB2 日志的原理

DB2 采用的是一种预写式日志（Write-Ahead Logging）方法来实现事务处理，简称 WAL。预写式日志的中心思想是：在数据库中更新、插入或删除的数据，被写入到磁盘之前，记录这

些变化的日志必须已经永久写入到磁盘。

首先分析一下图 8-6，预写式日志操作有个先后顺序，就是在数据缓冲区中即将被修改的数据，在被写入到数据磁盘之前，修改前的数据状态与修改后的数据状态都要先被写入到日志缓冲区，之后还要永久写入到日志文件中。这些操作完成后，才可以把修改后的数据写入数据磁盘中。这其中，控制把日志缓冲区中的日志记录写入日志文件的程序是写入程序（db2loggw）。

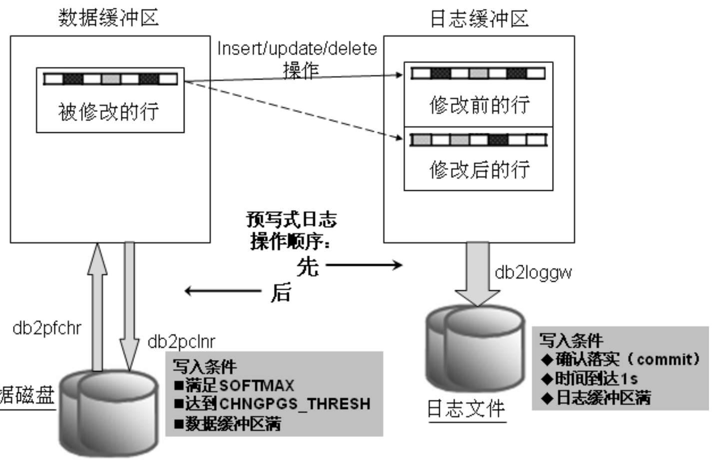


图 8-6　DB2 采用的预写式日志

另外，与数据库缓冲区关联的缓冲池预取程序（db2pfchr）和缓冲区页清除程序（db2pclnr）分别向缓冲区中读入数据和向磁盘中写入数据。

预写式日志的规则是日志为先，数据在后。下面讨论一下这二者之间的关系。

对 DB2 日志来说，每一条日志记录首先会写入到日志缓冲区，再从日志缓冲区写入到磁盘。那么在什么条件下 DB2 会将日志缓冲区中的日志写入到磁盘呢？

- 当前事务被落实（Commit）时，此事务的日志必须由日志缓冲区写入到磁盘。
- 为了最小化数据丢失，每隔 1 秒，DB2 会将日志缓冲区的日志写入到磁盘。
- 在特殊情况下，当日志缓冲区满了，缓冲区中的日志会被立即写入到磁盘。
- 当数据缓冲区满时，如果置换出来的脏数据页需要写回磁盘，则在此之前，这些修改数据对应的日志记录必须先写回磁盘。

对于 DB2 数据缓冲区中的数据，数据缓冲区的数据写入到磁盘（也称为数据持久化）的次数要远远小于日志缓冲区的写入次数。那么在什么条件下 DB2 会将数据缓冲区中的数据写入到磁盘呢？

- 当 SOFTMAX 值达到限定值时，数据缓冲区中的数据会被写入到磁盘中。SOFTMAX 值是一个数据库配置参数，它决定了发生灾难恢复时所需要读取的日志文件的数目同时也决定了数据缓冲区向磁盘写入数据的频率。
- 影响数据写入磁盘频率的另一个参数是 CHNGPGS_THRESH，它标志着数据缓冲区中的被改变数据页的百分比，一般为 60%。即当数据缓冲区中的数据页有 60%以上被修改时，数据缓冲区会执行一次把数据写入磁盘的操作。
- 特殊情况下，当数据缓冲区满时，与日志缓冲区一样，数据缓冲区会立即执行把数据写入磁盘的操作。

注意：SOFTMAX 参数大小有何影响

SOFTMAX 参数决定了 DB2 在灾难恢复时需要读取日志文件的数目，同时它也决定了数据缓冲区向磁盘写入数据的频率。当把 SOFTMAX 值设置较小时，当发生灾难恢复时的恢复速度会变快，但是数据缓冲区写数据的频率会变高，即 I/O 次数会变大。反之，如果采用较大的 SOFTMAX 值，则数据库 I/O 频率会降低，但是一旦发生灾难恢复时，DB2 进行恢复的时间会延长。一般情况下，设置 SOFTMAX 值区间为 100~300。

那么 DB2 的日志文件到底什么样呢？先看看日志文件的真面目吧。

```
[db2inst1@db2 ~]$ db2 get db cfg for SAMPLE | grep "Path to log files"
Path to log files = /db2/db2lde/data_sampl/db2lde/NODE0000/SQL00001/SQLOGDIR/

[db2inst1@db2 ~]$ cd /db2/db2lde/data_sampl/db2lde/NODE0000/SQL00001/SQLOGDIR/

[db2inst1@db2 ~]$ ls -lrt
total 2.6G
-rw------- 1 db2lde dbldeadm 512 2009-05-27 18:52 SQLLPATH.TAG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004717.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004716.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004715.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004714.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004713.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004712.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004711.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004710.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004709.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004708.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004707.LOG
-rw------- 1 db2lde dbldeadm 64M 2011-11-05 00:00 S0004706.LOG
......
```

参照上面的示例，读者需要了解的是：

- DB2 日志是以文件形式存储在存储介质的，如日志中的 S0004717.LOG 文件。
- DB2 日志文件的名字是以 Sxxxxxxx.LOG 形式存在，其中 xxxxxxx 为 7 位数字，从 0000000 到 9999999，即日志可从 S0000000.LOG 排到 S9999999.LOG。
- 每一个日志文件的大小一致，如示例中的大小为 64M，其大小可以通过数据库配置参数 LOGFILSIZ 调整。
- DB2 日志文件是以二进制形式存储，它需要特殊的工具才可以查看。
- 日志文件会存储在指定的日志目录中，由数据库级参数（Path to log files）指定。如上例中为/db2/db2lde/data_sampl/db2lde/NODE0000/SQL00001/SQLOGDIR，它的位置可以通过数据库配置参数 NEWLOGPATH 调整。

实际上，在 DB2 运行的过程中，一个日志文件中会包含多个事务的修改记录。当前日志文件被写满后，DB2 会将日志文件序号自动加 1，并写入到新的文件中。

如图 8-7 所示，DB2 在一段时间内同时处理 6 个事务 A、B、C、D、E 和 F。这些事务首先会被记录在日志缓冲区中，当写入磁盘的条件满足时，这些日志记录会被写入到磁盘上的日志文件中，一个日志文件中会保存多个不同事务的变化记录。

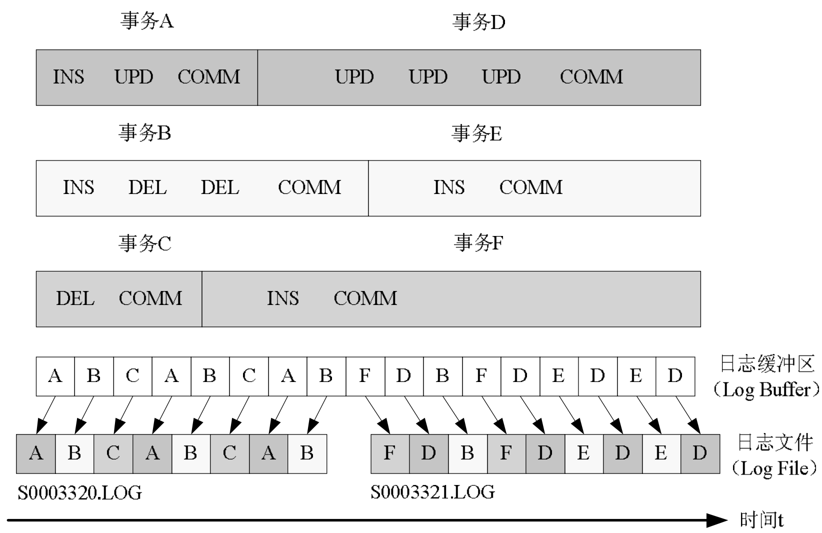


图 8-7　日志文件的使用

8.4.2　DB2 日志的模式

DB2 中日志运行模式分为两种，循环日志模式（Circular Logging）和归档日志模式（Archival Logging）。

需要注意的是，无论循环日志模式还是归档日志模式，都包含主日志文件（Primary Log）和辅助日志文件（Secondary Log）。

- 主日志文件是记录日志的主要文件，在 DB2 启动的时候即进行分配，其数目由参数 LOGPRIMARY 确定。
- 辅助日志文件是当主日志文件已满时，为保证当前事务可以被继续处理，按照需要分配一个新的日志文件，直到分配的日志数达到规定的最大数目。如果这时还不能完成当前事务，则当前事务会被撤销并返回错误代码至上层应用程序。辅助日志参数由 LOGSECOND 确定。

在数据库配置参数中可以查看或修改主日志个数（LOGPRIMARY）、辅助日志个数（LOGSECOND）和日志文件大小（LOGFILSIZ）。对于 DBA 来说，可以通过如下的 3 个参数值，计算出 DB2 支持的最大日志空间。

```
最大日志空间 = LOGFILSIZ ×（LOGPRIMARY + LOGSECOND）
```

在下面的示例中，最大日志空间为 16380×4KB×（40 + 60）= 6.4 GB。在最新的 DB2 V9.7 版本中，日志文件最大上限为 1024GB。

```
Log file size (4KB)                        (LOGFILSIZ) = 16380
Number of primary log files               (LOGPRIMARY) = 40
Number of secondary log files              (LOGSECOND) = 60
Changed path to log files                 (NEWLOGPATH) =
Overflow log path                    (OVERFLOWLOGPATH) =
Mirror log path                        (MIRRORLOGPATH) =
First active log file                                  = S0004695.LOG
```

```
First log archive method              (LOGARCHMETH1) = DISK:/db2/LDE/log_archive/
 Options for logarchmeth1              (LOGARCHOPT1) =
Second log archive method             (LOGARCHMETH2) = OFF
isis:db2lde 73>
```

弄清楚了主日志和辅助日志的区别，下面看一下不同日志模式的区别。

1. 循环日志模式

循环日志模式是默认的日志使用方式。顾名思义，循环日志的工作模式指的是，数据库会向设定好的几个日志文件中依次写入日志记录，并循环往复，它会覆盖掉以前使用过的文件。

如图 8-8 所示，当前的日志记录会写入到主日志 2 中，当主日志 2 写满后，DB2 会依次向主日志 3 和主日志 4 中写入。若均已写满，则 DB2 会继续使用主日志文件 1 并覆盖掉之前的内容。

另外一种情况，假设当前有一个持续时间很长的事务，将主日志 2、主日志 3 和主日志 4 均已写满，但是此事务还没有结束（还未 Commit），则 DB2 会自动分配一个辅助日志 1，继续处理。如果事务还未结束，DB2 会继续分配辅助日志 2、辅日志 3 直到辅助日志最大值。这样，此事务或者在日志用尽后被回滚，或者在之前成功完成。

注意：

循环日志模式仅支持灾难恢复，数据库离线备份和版本恢复等恢复方式。

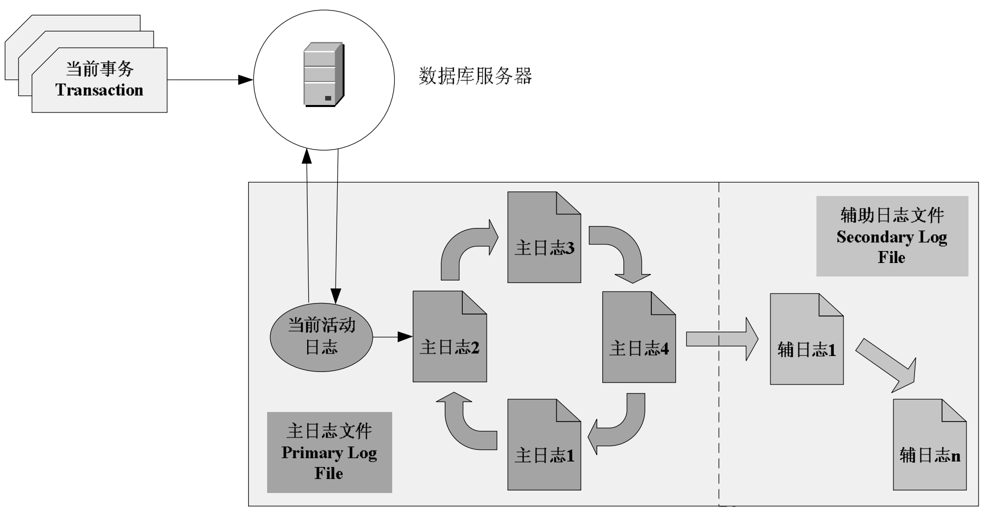


图 8-8　DB2 循环日志模式

2. 归档日志模式

归档日志模式是生产环境下的常用模式。如图 8-9 所示，它也是使用 4 个主日志循环写入。所不同的是，在归档模式下，当企图覆盖以前的日志文件时，DB2 会将此日志文件归档，从而避免被覆盖。一般情况下，需要在生产系统或主要的业务系统中开启归档日志模式。那么，使用归档日志模式能为数据库管理带来什么好处呢？

首先，使用归档日志模式可以让你使用归档日志文件，将数据前滚（Rollforward）到任意时间点，这保证了数据库中的数据在任意时刻是可恢复的。另外，开启归档日志模式后，可以使用在线备份（Online Backup）。

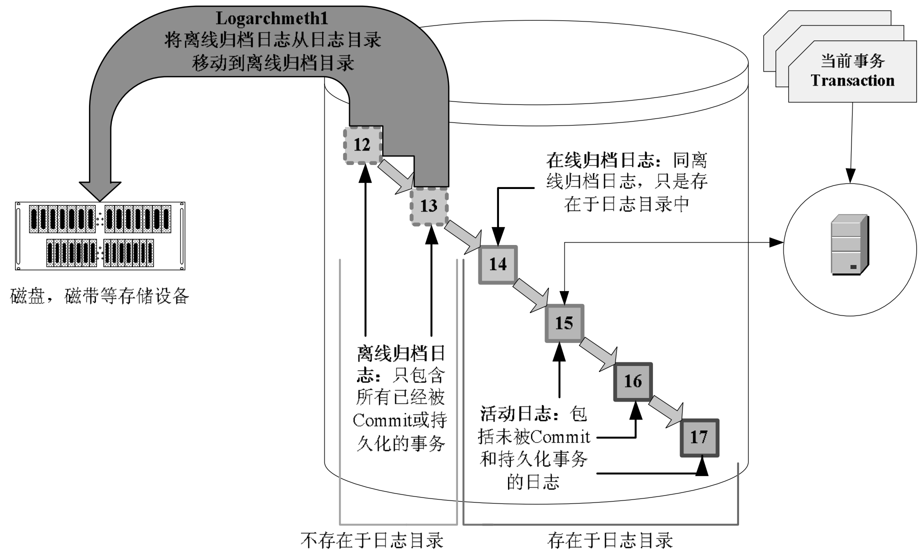


图 8-9　DB2 归档日志模式

需要注意的是，如果需要开启高可用性和灾备特性（HADR）或使用日志传输（Log Shipping）功能，也必须开启归档模式。

在开启归档模式后，需要监控产生的日志文件是否被正确归档到磁盘或者 TSM 等设备上。否则，大量的归档日志会充满日志文件夹，从而导致数据库系统立即停机。

此外关于归档日志，有下面几个容易混淆的概念，需要大家理解清楚。

（1）活动日志

数据库正在往其中写入日志数据，或者其中包括还没有提交（Commit）记录的那些日志文件。如果有一个持续时间很长的事务使得所有的日志均为活动日志，则为了保证此事务的正常完成，则数据库会依次分配辅助日志。注意，活动日志是不能被归档的。

（2）在线归档日志

在线归档日志中记录的所有事务均是被提交的，但这个日志又没有从日志目录中转移到归档目录中，所以称它为在线归档日志。当日志循环企图覆盖在线归档日志时，在线归档日志会被转移至离线归档目录变为离线归档日志。

（3）离线归档日志

离线归档日志只包括已提交事务日志，并且已转移到离线归档目录下。一般来讲，只有在做数据库前滚恢复时才会使用离线归档日志。

注意：

归档日志模式是生产环境下的推荐模式，它支持灾难恢复、数据库在线备份、版本恢复、前滚恢复以及高可用性和灾备（HADR）等。

8.4.3　日志优化的最佳实践

当 iostat 或者 perfmon 监控工具发现日志磁盘的平均 I/O 过大且速度小于每秒 50 次 I/O 读写，说明出现了日志瓶颈。要想解决日志瓶颈，可以重新调整存储设计来解决，例如日志和数据所用磁盘物理隔离、增加日志磁盘个数等。

但是很多时候，由于各种条件的限制，不允许 DBA 重新调整存储，所以只能减少日志量，这包括下面的手段：

- 对拥有 BLOB 和 CLOB 等大对象字段的表使用 NOT LOGGED 选项，这样在对大对象进行插入、删除和修改时，不会产生大量的日志。
- 在执行 ALTER TABLE 操作时，可能整个表的数据都会被修改，并被记录日志。这时候可以使用 NOT LOGGED INITIALLY 选项来减少日志量。
- 对临时表使用 NOT LOGGED 选项，这样就不会记录日志了。
- 在大批量导入数据的时候，尽量使用 LOAD 工具，这是因为 LOAD 工具加载数据时不会通过 DB2 引擎，既高效又不会记录日志。
- 减少 COMMIT 的次数。每次执行 COMMIT 操作都会写日志。如果可能，应该修改应用程序，减少 COMMIT 次数，使一次 COMMIT 所提交的修改比较多，减少日志写入磁盘的次数。
- 增加数据库参数 MINCOMMIT 的值。MINCOMMIT 能够使数据库延迟提交事务，直到事务提交的请求达到这个参数指定的数目。
- 使用 TRUNCATE 命令来清空整个表。TRUNCATE 比 DELETE 速度快，而且记录的日志量少。
- 使用压缩功能。当对表启用压缩时，日志记录的数据也会被压缩，这样就减少了日志的 I/O 吞吐量。

8.5 并发机制比较：DB2 vs Oracle

DB2 的并发机制与 Oracle 的并发机制存在很大区别。DB2 和 Oracle 在并发机制上不同的根本原因在于锁与日志的实现机制不同。下面来比较一下 DB2 和 Oracle 在锁与日志上的区别。

8.5.1 锁与隔离级别：DB2 vs Oracle

DB2 的锁和隔离级别与 Oracle 的实现机制存在很大区别。总结起来，这些区别主要体现在以下几个方面：

（1）DB2 在做数据修改时，在日志中既记录了修改前数据（也就是 UNDO 日志），也记录了修改后的数据（即 REDO 日志）。而 Oracle 只在日志中记录修改后数据，而将修改前的数据记录到“回滚段”（ROLLBACK SEGMENT）中。

（2）在 Oracle 中，当一个应用程序对表以 Insert、Update 和 Delete 操作进行修改时，另外一个应用程序在读取该表时，会从回滚段中读取该表最近修改前的数据。而对于 DB2 V9.7 之前的版本，读取数据的应用程序，遇到正在被其他应用程序修改的数据时，将会进行锁等待（除非使用 UR 隔离级）。对于 DB2 V9.7 及以后版本，由于引入了当前已落实，读取数据的应用程序将不需要等待锁释放，而是会从日志中读取数据修改前的版本（即已落实的版本，相当于 Oracle 回滚段中的数据）。

（3）Oracle 利用数据行上的标志位来实现锁机制，而 DB2 是通过把锁信息记录在锁列表（LOCKLIST）中来实现的。当请求加锁时，DB2 会检查锁列表，看数据对象上是否已加锁，以及请求的锁与已加的锁是否兼容。

（4）Oracle 数据库中，对数据的读操作是不加锁的。而 DB2 强调“读一致性”，在读数据行时，会根据隔离级的不同而加 S 或 IS 锁，只有在使用 UR 隔离级时才不加 S 或 IS 锁，这保证了不同应用程序和用户读取的数据是一致的。默认情况下，Oracle 同一时刻不同的应用程序有读不一致的现象，而 DB2 在同一时刻所有的应用程序读到的数据都是一致的。

（5）Oracle 数据库中不存在锁升级，而 DB2 数据库中当锁列表（LOCKLIST）被耗尽，或者单个事务锁占用的内存空间超过 LOCKLIST×（MAXLOCKS/100）个页时会发生锁升级。

（6）DB2 缺省的隔离级是 CS，对于大多数应用来说，缺省的 CS 级别可以满足需要。Oracle 的默认隔离级 read committed 和 CS 类似，通过多版本机制，不会出现脏读现象。

（7）DB2 在 UR 隔离级下，遇到正在被更改的数据会读最新的没有提交的脏数据；而 Oracle 在其所有的隔离级（即 read committed、serializable 以及 read-only）下都不会读取脏数据（在 read committed 级别下，会从回滚段中直接读取修改前的曾经提交过的数据）。在一些应用中，需要读到最新的数据，也就是脏数据。对这两种方法严格来说并没有谁好谁坏，ANSI 标准也并没有明确规定。

8.5.2　日志：DB2 vs Oracle

8.4 节介绍了 DB2 日志的原理和模式，那么 DB2 的日志与 Oracle 的日志有何区别？它们又分别是如何运行和管理的呢？

DB2 在做数据修改时，在日志中既记录了修改前的数据（也就是 UNDO 日志），也记录了修改后的数据（即 REDO 日志）。而 Oracle 只在日志中记录修改后数据，而将修改前的数据记录到“回滚段”（ROLLBACK SEGMENT）中。

Oracle 数据库是通过重做日志（Redo Log）和重做日志缓冲区（Redo Log Buffer）来对数据库日志进行管理。同时，它还可以使用 3 个冗余的重做日志组（Redo Log Group）来保证在线日志的安全。

在 Oracle 中，其在线日志的运行模式又分为 2 种：非归档模式（NOARCHIVELOG）和归档模式（ARCHIVELOG）。非归档模式指的是在线日志在多个重做日志组中循环使用。归档模式是指多个重做日志组循环使用，当所有的日志组被用尽，回到初始第一个日志组要进行覆盖前，此日志组中的内容会被归档到其他位置。

实际上，DB2 与 Oracle 在日志管理上有不少相同点。熟悉 Oracle 的朋友可以体会到，几乎所有在 Oracle 中与日志有关的概念，都可以直接在 DB2 中找到对应的对象。不过，与 DB2 不同的是，在 Oracle 中没有主日志文件和辅助日志文件这样的区别。

比较：Oracle 与 DB2 日志术语对比

它们之间的对比，如表 8-6 所示。

表 8-6　Oracle 与 DB2 日志术语对比

Oracle	DB2
重做日志文件（Redo Log File）	日志文件（Log File）
重做日志缓冲区（Redo Log Buffer）	日志缓冲区（Log Buffer）
重做日志组（Redo Log Group）	镜像日志路径（Mirrorlogpath）
非归档模式（NOARCHIVELOG）	循环日志模式（Circular Logging）
归档模式（ARCHIVELOG）	归档日志模式（Archival Logging）

对于并发控制，无论是 DB2 还是 Oracle，都有自己的特点，DBA 需要根据它们不同的特点合理设计应用并配置好相关参数。

8.6　并发性能优化

锁与日志在 DB2 中都是资源，它们是并发控制的关键，要优化锁与日志的利用效率，要灵活运用各种策略，精打细算，想办法使系统并发性最大化，下面就从应用程序和数据库两方面出发，探讨如何优化并发性能。

8.6.1　应用开发

系统的并发性能很大程度上取决于应用的好坏。如果应用程序对 DB2 的并发机制使用不当，比如事务未及时提交、使用不当的隔离级别等，那么会产生各种并发问题。

1. 编写高效 SQL 语句

SQL 语言非常灵活，可以使用多种写法实现同一功能，但是最高效的只有一种。本书第 9 章 9.5 小节深入探讨了如何编写高效 SQL 语句，读者可以学习一下。

2. 创建合适索引

创建合适索引，优化器就可以通过索引扫描只访问符合索引条件的行，并对它们加行锁，这样就减少了加锁数量；如果使用限定数据范围的查询条件（Range Predicate），将会进一步减少行锁。但是索引过多会导致插入、更新和删除操作的性能，索引不是越多越好。一般来说，OLTP 系统，一个表上的索引 3~5 个比较合适，OLAP 系统，5~8 个比较合适。

3. 编写高效事务

在应用程序中，要编写高效事务：

（1）在应用逻辑允许的情况下尽早使用 COMMIT 语句来提交事务，以减少持有锁的时间，并释放日志资源。应用程序发生错误时，也应该及时回滚事务，释放相关锁与日志资源。

（2）尽量不要在事务中加入执行时间过长或不必要的代码，比如大的代码循环和远程调

用，如果有这些逻辑，尽量放在读取数据或者更新数据之前，或者事务提交以后，这样就能减少锁的占有时间。

（3）在事务中应该尽量晚地访问关键数据，也就是尽量把对关键数据的访问放到事务即将提交时，这样能最大限度的减少锁定关键数据的时间。

（4）如果多个应用程序访问相同的一系列资源，应该按相同的顺序依次访问，这样可以避免了死锁的发生。

（5）尽可能快地关闭游标。因为使用游标时会锁定游标当前指向的数据，而关闭游标，可以释放锁。

4. 灵活运用 DB2 提供的高级设计技术

在第 5 章谈到表分区、MDC、MQT 等高级技术，灵活运用它们，可以取得事半功倍的效果。

8.6.2　数据库调优

在应用程序开发的时候，使用上述建议能够减少将来出现并发问题的可能性。但是，这还不够，还需要从数据库层面进行必要的调优工作。

1. 设置合适的隔离级

从 8.2.1 节了解到，DB2 提供了四种隔离级，按照严格程度，从高到底为，分别为 RR、RS、CS 和 UR。隔离级越高，数据稳定性越好，但是并发性能越差。

为了设定合适的隔离级，首先需要确定都运行了哪些应用程序，这些应用程对数据稳定性和高并发性有什么样的需求，随后根据这些需求在不同层次选择不同的隔离级。

例如，在笔者的一次调优工作中，将一些报表应用的隔离级由 CS 改为 UR，解决了锁等待，使得平均响应时间从几分钟降到几秒，直接增加了系统的吞吐量，避免了为了支持更多用户而升级系统硬件。

如果应用逻辑允许，尽量激活 DB2 V9.7 提供的当前已落实特性，这能有效提升数据库的并发性。

2. 设置合理的锁参数和日志参数

锁参数包括 LOCKLIST、MAXLOCKS、DLCHKTIME 和 LOCKTIMEOUT，日志参数包括 LOGFILSIZ、LOGPRIMARY、LOGSECOND 和 LOGBUFSZ，它们的设定参见第 7 章 7.1.3 小节。在高并发情况下，这些参数如果设置不当会形成瓶颈，严重影响并发性。从最大并发性出发，要设置好这些参数。

3. 优化关键 SQL 语句

占用资源多、运行频率高的关键 SQL 语句对并发性影响很大，它们通常占用了大量锁资源（长时间持有锁、占用大量锁内存等），要通过分析访问计划，对这些关键 SQL 语句进行优化，从而降低它们因为持有锁所产生的资源消耗。关于具体的优化方法，请参考本书第 9 章的有关内容。

8.7　小结

本章对 DB2 并发控制进行了详尽地阐述。首先从整体上介绍了学习和掌握 DB2 并发机制的方法。接着介绍了并发机制中的事务、锁和隔离级的原理和相关例子。然后介绍了并发问题的监控和分析方法，对锁等待、锁超时和死锁进行了全面的分解与阐述。最后介绍了如何实现 DB2 并发性的最大化，提出了诸多实际操作建议和规范。

锁是保证数据一致性的机制，而日志是保护数据的最基本策略。本章还介绍了 DB2 日志原理和模式，并和 Oracle 进行了比较。

通过本章的学习，读者能够深入理解 DB2 的并发机制和日志原理，活学活用，在系统设计和性能调优工作中，能游刃有余地处理各种并发问题。

第 9 章 SQL 语句优化实战

SQL 语句优化至关重要，这是因为它对性能影响大，例如很多性能瓶颈都是由耗费资源的关键 SQL 语句（Costly SQL）导致的。

特别重要的是，SQL 语句优化持续周期长，往往贯穿设计、开发和运行维护整个生命周期：

- 设计阶段：设计人员不仅要关注数据模型，还要确保表和索引的优良设计，从而确保 SQL 语句的高效执行。
- 开发阶段：很多开发人员编写的 SQL 语句性能低下，需要优化其中的关键 SQL，以满足业务的性能需求。
- 运行维护阶段：随着数据本身的变化特别是数据量的大幅增加，数据库的性能开始变慢，DBA 需要优化某些关键 SQL 来提升性能。

SQL 优化的前提是通过监控发现问题 SQL 语句，读者可以参考第 6 章 6.4 节的相关内容。找到关键的问题 SQL 语句后，就可以对它们采取针对性的优化方案了。

但是，SQL 语句优化不是一件容易的事，需要掌握正确的方法，即深入理解 DB2 优化器，掌握分析和调整访问计划的方法，并加以不断地实践，才能逐渐掌握 SQL 语句优化的技能。

“授人以鱼不如授人以渔”，本章将深入讲解 SQL 语句优化原理和最佳实践。

9.1 SQL 优化基础，理解 DB2 优化器

优化器是 DB2 的大脑，SQL 语句在 DB2 引擎中执行以前，首先需要优化器为其生成访问计划。正如在战争中，错误的作战计划可能导致全军覆没一样，错误的访问计划也会导致 SQL 语句执行效率低下甚至拖累整个系统。

本节将首先讲解 SQL 语句的编译过程，随后探讨 DB2 优化器的工作原理。

9.1.1 SQL 语句的编译过程

如图 9-1 所示，SQL 语句的编译过程需要经过下面几个步骤：

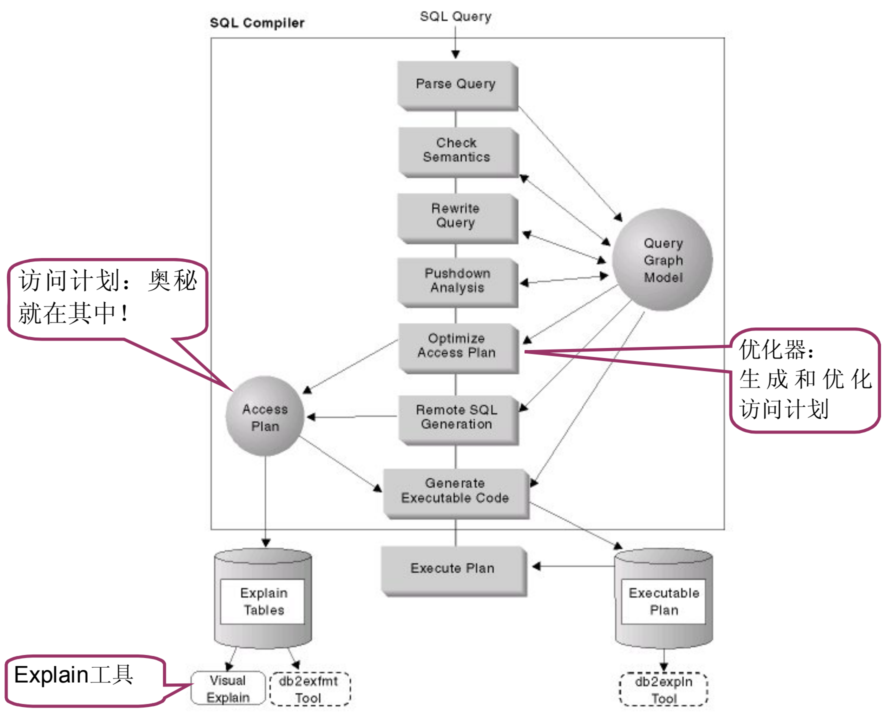


图 9-1　SQL 语句编译过程

（1）分析语法（Parse Query）：编译器从语法层面分析 SQL 语句，如果检测到任何语法错误，就会停止处理，返回相应的错误提示给用户。经过语法分析后，SQL 语句就被转换成内部表示形式，即 QGM（Query Graph Model），QGM 用图结构来表示 SQL 语句，后续的编译过程将围绕 QGM 展开。

（2）检查语义（Check Semantics）：编译器从语义层面分析 SQL 语句，例如类型检查、参照完整性约束、触发器和检查约束等，语义分析后的对象也加到 QGM 中。

（3）重写查询（Rewrite Query）：在不改变语义的基础上，对查询进行等价变换。例如，将谓词下推到基本表作为局部谓词；用 MQT 重写查询；将子查询转换成连接。

例如，下面的 SQL 语句使用了 IN 子查询：

```
SELECT EMPNO, FIRSTNME, LASTNAME, PHONENO
FROM EMPLOYEE
WHERE WORKDEPT IN (SELECT DEPTNO FROM DEPARTMENT WHERE DEPTNAME = 'OPERATIONS')
```

IN 子查询被转换成连接，以便于优化器考虑各种连接方法：

```
SELECT DISTINCT EMPNO, FIRSTNME, LASTNAME, PHONENO
FROM EMPLOYEE EMP,DEPARTMENT DEPT
WHERE EMP.WORKDEPT = DEPT.DEPTNO AND DEPT.DEPTNAME = 'OPERATIONS'
```

（4）下推分析（Pushdown Analysis）：处理联邦查询（Federated Query）独有的步骤，其主要任务是确定查询中的哪些操作可以放到远程数据源执行。

（5）优化访问计划（Optimize Access Plan）：优化器分析 QGM，根据表和索引上的统计信息，并参考当前数据库的各种参数配置，估计各种访问计划的成本，然后选择成本最小的计划作为最终的访问计划。

（6）远程 SQL 生成（Remote SQL Generation）。处理联邦查询时独有的步骤，编译器会根据远程表的访问计划，生成用于在远程数据源执行的 SQL 语句。

（7）生成可执行代码（Generate Executable Code）：编译器根据访问计划和 QGM 生成一个可由 DB2 引擎执行的二进制代码段（Section）供 DB2 运行时使用。这个代码段不可读，其中包含的访问计划信息可以通过工具 db2expln 查看。

9.1.2　DB2 优化器的核心作用

在 SQL 编译过程中，最核心的步骤便是生成和优化访问计划。下面探讨一下优化器的作用，如图 9-2 所示，包含下面的要点：

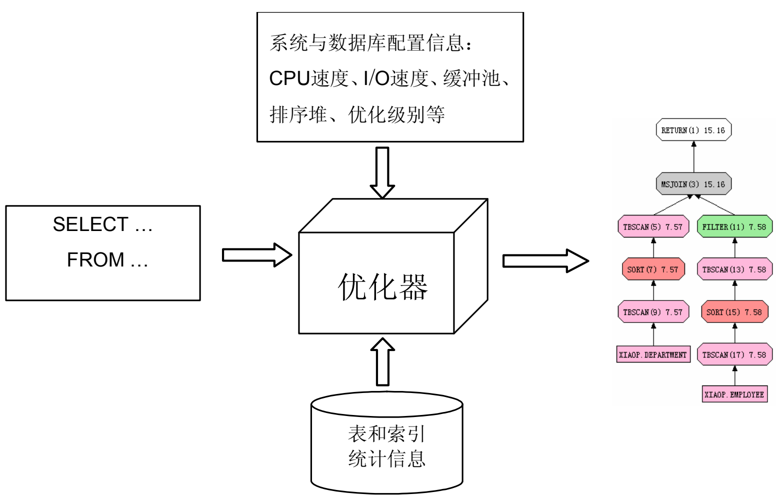


图 9-2　DB2 优化器的核心作用

（1）以 SQL 语句作为输入：优化器将重点考虑 SQL 中谓词的处理效率和表的连接方式。

（2）系统与数据库的设置：优化器会充分考虑 CPU 速度、存储的 I/O 速度、并行模式、并发环境、优化级别、缓冲池和排序堆大小等。

（3）表和索引统计信息：统计信息是优化器进行 SQL 语句优化的基础。

（4）生成最优访问计划：优化器充分考虑表访问方式、连接方法以及操作顺序，生成各种可能的访问计划，并根据成本模型计算每个访问计划的成本，最后选择一个总成本最小的访问计划。

9.1.3 SQL 语句优化过程

了解 SQL 语句的编译过程和 DB2 优化器原理后，理解 SQL 语句的优化过程就很容易了，一般包括下面四个步骤：

（1）通过性能监控，发现存在性能问题的关键 SQL 语句。关于性能监控的方法请参考第 6 章。

（2）分析 SQL 语句的访问计划，找到问题的原因所在，比如索引缺失、数据库参数设置不合理、统计信息不准确或者更新不及时等。

（3）调整 SQL 语句的访问计划，例如添加索引、调整数据库参数、修改 SQL 语句等。

（4）进行性能验证，确保调优措施有效。

在这四个步骤中，关键点是分析和调整 SQL 语句的访问计划。

9.2 SQL 优化关键：分析访问计划

上一节我们探讨了 DB2 优化器，这是掌握 SQL 语句优化的理论基础。本节将深入探讨 SQL 语句优化的关键点，即如何使用工具生成关键 SQL 语句的访问计划，如何分析访问计划，找到性能问题的原因所在，从而找到优化关键 SQL 语句的办法。最后介绍使用优化概要定制访问计划的方法，这也是 SQL 优化的有力武器。

9.2.1 解释（Explain）工具

DB2 的解释工具（Explain Tool）能打印出 SQL 语句的访问计划，从而用于分析以确认性能问题所在。为了查看访问计划，DB2 提供了三种常用工具：Visual Explain、db2exfmt 和 db2expln，下面分别介绍。

比较：Explain 工具的区别

- db2exfmt：命令行工具，将访问计划输出为文本形式，包含最详尽的信息。
- db2expln：命令行工具，打印出代码段（section）中的访问计划信息，并可用于打印已有存储过程中静态 SQL 语句的访问计划。
- Visual Explain：图形化的 Explain 工具，控制中心、Data Studio、Toad 等都支持图形化的解释工具。

---

1. db2exfmt

db2exfmt 是最常用的命令行解释工具，它在各种平台通用，而且它输出的访问计划包含详细的优化器信息。在一些安全管理比较严格的客户那里，只能使用命令行或者不让安装使用图形化工具，这时 db2exfmt 就能派上用场。

使用 db2exfmt 工具前，需要创建相关的解释（explain）表。先切换到目录 sqllib/misc（windows 平台为 SQLLIB\MISC）下，然后执行 EXPLAIN.DDL 脚本，用于创建一组相关联的表，如下所示。

```
db2 -tvf EXPLAIN.DDL
```

创建解释表的另一种方式是调用下面的存储过程，其中第二个参数“C”的含义为创建。相应的，如果传入参数“D”，则表示删除这些 explain 表。

```
call sysproc.sysinstallobjects('EXPLAIN','C','',CURRENT SCHEMA);
```

用 db2exfmt 获取 SQL 访问计划的步骤很简单，它的运行方式如下所示。在 explain 模式下，DB2 只编译而不会执行查询，编译得到的访问计划写入到解释表中，然后 db2exfmt 从解释表中读取相关内容，并格式化输出成用户可读的访问计划。

```
db2 explain plan for select ...;
db2exfmt  -d <db_name> -1 -o <outputfile>
或者
db2 set current explain mode explain
select ...
 db2 set current explain mode no
 db2exfmt -d <db_name> -1 -o <outputfile>
```

注意：解释含参数标记的动态 SQL

含有参数标记的 SQL 语句也可以用 db2exfmt 生成其访问计划。如 explain plan for select C_CUSTOMER_SK from tpcds.customer where C_BIRTH_YEAR = ?

小技巧：清除 explain 表中的旧数据

DB2 在解释查询时只会向 explain 表里面插入新数据，不会删除旧数据。随着解释 SQL 语句的次数越来越多，解释表会越来越大，这时可以用下面的命令把所有的解释数据删除：

```
delete from EXPLAIN_INSTANCE
```

2. db2expln

与 db2exfmt 不同，db2expln 从代码段中打印出访问计划（参考图 9-1），它不需要创建解释表，使用起来更加方便。

使用 db2expln 命令的语法如下所示，其中选项-statement 指定需要解释的 SQL 语句，选项-g 指定打印出树形的访问计划，-o 指定输出文件。

```
    db2expln -d tpcds -statement "select C_CUSTOMER_SK from tpcds.customer where
C_BIRTH_YEAR = ?" -g -o expln.output
```

也可以将 SQL 语句放在文件（比如 statements.sql）中，这时 db2expln 的命令如下，其中 -stmtfile 指定包含 SQL 语句的文件名：

```
    db2expln -d tpcds  -stmtfile statements.sql -g -o expln.output
```

输出的访问计划主要内容如下所示，前半部分用层次结构表示访问计划，后半部分树形访问计划图则和 db2exfmt 命令的输出类似，不过它不会包含每个操作符的详细信息。注意加黑的 **"ID = 2,6"**，其中 2 为表空间 ID， 6 为表 ID。

```
    Section Code Page = 1208

    Estimated Cost = 15.473010
    Estimated Cardinality = 74.000000

    Access Table Name = DB2INST1.EMPLOYEE  ID = 2,6
    |  #Columns = 3
    |  Skip Inserted Rows
    |  Avoid Locking Committed Data
    |  Currently Committed for Cursor Stability
    |  May participate in Scan Sharing structures
    |  Scan may start anywhere and wrap, for completion
    |  Fast scan, for purposes of scan sharing management
    |  Scan can be throttled in scan sharing management
    |  Relation Scan
    |  |  Prefetch: Eligible
    |  Lock Intents
    |  |  Table: Intent Share
    |  |  Row  : Next Key Share
    |  Sargable Predicate(s)
    |  |  Process Build Table for Hash Join
    Hash Join
    |  Estimated Build Size: 4000
    |  Estimated Probe Size: 8000
    |  Access Table Name = DB2INST1.DEPARTMENT  ID = 2,5
    |  |  #Columns = 2
    |  |  Skip Inserted Rows
    |  |  Avoid Locking Committed Data
    |  |  Currently Committed for Cursor Stability
    |  |  May participate in Scan Sharing structures
    |  |  Scan may start anywhere and wrap, for completion
    |  |  Fast scan, for purposes of scan sharing management
    |  |  Scan can be throttled in scan sharing management
    |  |  Relation Scan
    |  |  |  Prefetch: Eligible
    |  |  Lock Intents
    |  |  |  Table: Intent Share
    |  |  |  Row  : Next Key Share
    |  |  Sargable Predicate(s)
    |  |  |  Process Probe Table for Hash Join
```

```
Return Data to Application
|  #Columns = 3

End of section


Optimizer Plan:

          Rows
        Operator
          (ID)
          Cost

           74
          n/a
         RETURN
          ( 1)
         15.473
           |
           74
          n/a
         HSJOIN
          ( 2)
         15.473
        /      \
     102       74
     n/a      n/a
   TBSCAN    TBSCAN
    ( 3)      ( 4)
   7.73946   7.7015
     |         |
    102        74
    n/a       n/a
 Table:     Table:
 DB2INST1       DB2INST1
 DEPARTMENT  EMPLOYEE
```

尽管 db2expln 的输出没有 db2exfmt 详尽，但它却有独特的用武之地：打印出已创建的 SQL 存储过程（或者程序包）的访问计划，这对分析存储过程的性能相当有用。

下面的例子用 db2expln 获取 SQL 存储过程 DB2INST1.SP 的访问计划：先从系统编目中找到相应的程序包名，然后用 db2expln 打印出该程序包的访问计划。

```
首先根据存储过程名字找到程序包名，这里为 P7544528
db2 "select r.ROUTINESCHEMA, r.ROUTINENAME, d.BNAME, d.BSCHEMA
     from syscat.routines r, syscat.routinedep d
     where r.ROUTINESCHEMA='DB2INST1' and r.ROUTINENAME='SP'
       and r.SPECIFICNAME = d.SPECIFICNAME and d.BTYPE ='K'"
接着，打印出程序包 P7544528 的访问计划
db2expln -d tpcds -c DB2INST1 -p P7544528 -g -o sp.expln
```

注意：SQL 存储过程的访问计划

SQL 存储过程的访问计划可以通过 db2expln 获取，如果要用 db2exfmt 抓取更为详尽的访问计划，这就需要在创建存储过程之前设置 EXPLAIN YES 这个编译选项了。有两种方法来设置这个选项：

- 调用存储过程 SET_ROUTINE_OPTS 设置 EXPLAIN YES 编译选项，于是在当前会话中创建的 SQL 存储过程，DB2 会将其访问计划写入到解释表中，从而可以用 db2exfmt 工具获取。在每次 rebind 的时候，该存储过程中的 SQL 语句会重新编译，其访问计划也将重新被写入到解释表。

```
CALL SET_ROUTINE_OPTS('EXPLAIN YES')
```

- 设置注册变量 DB2_SQLROUTINE_PREPOPTS，这需要重新启动 DB2 实例才能生效，随后创建的 SQL 存储过程将能用 db2exfmt 抓取访问计划。

```
Db2set DB2_SQLROUTINE_PREPOPTS="EXPLAIN YES"
```

3. Visual explain

图形化的解释工具操作简单，显示的访问计划形象直观。用 DB2 控制中心的 Visual Explain 解释查询的效果如图 9-3 所示。它对访问计划图中的各种节点进行不同着色，直观清晰。

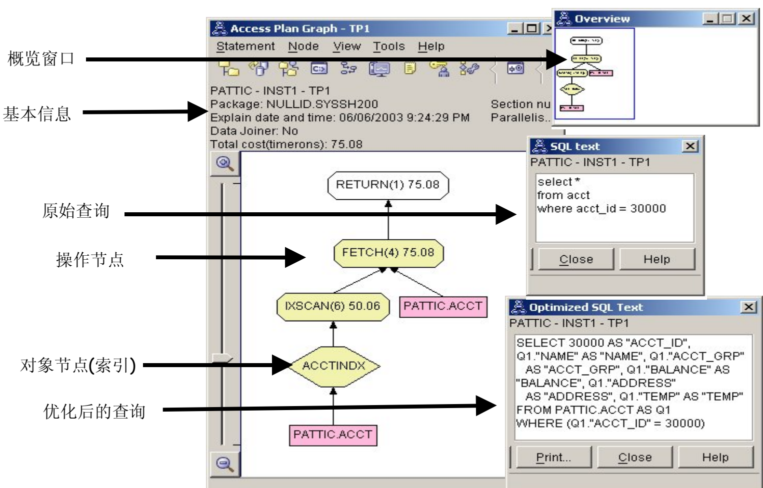


图 9-3　DB2 控制中心的 Visual Explain 示例

其他一些图形化工具如 Data Studio、Toad for DB2 等也能直接分析访问计划。

9.2.2　解读访问计划

本章的访问计划解读主要基于 db2exfmt 的输出，因为它包含最详细的优化器信息。

总体来说，db2exfmt 打印出的信息主要包含下面的部分，与图 9-2 中优化器的输入和输出相对应：

- 全局上下文信息：包括数据库配置、优化级别和隔离级别等。
- SQL 语句文本：包括原始 SQL 语句和优化后的 SQL 语句。

- 访问计划：包含访问计划图和每个操作符的细节信息，访问计划中用到的对象以及统计信息，包括表、索引等。

1. 全局上下文信息

在 db2exfmt 输出的开始部分，包含着丰富的上下文信息，这些信息直接影响访问计划的生成，在分析时不可忽视。

如下所示，一个访问计划的全局上下文信息：

```
******************** EXPLAIN INSTANCE ********************

DB2_VERSION:       09.07.4
SOURCE_NAME:       SQLC2H21
SOURCE_SCHEMA:     NULLID
SOURCE_VERSION:
EXPLAIN_TIME:      2013-02-28-11.35.17.327941
EXPLAIN_REQUESTER: DB2INST1

Database Context:
----------------
    Parallelism:          None
    CPU Speed:            5.707493e-07
    Comm Speed:           100
    Buffer Pool size:     13740
    Sort Heap size:       56
    Database Heap size:   2271
    Lock List size:       4096
    Maximum Lock List:    97
    Average Applications: 1
    Locks Available:      127139

Package Context:
---------------
    SQL Type:           Dynamic
    Optimization Level: 5
    Blocking:           Block All Cursors
    Isolation Level:    Cursor Stability

---------------- STATEMENT 1  SECTION 203 ----------------
    QUERYNO:       1
    QUERYTAG:
    Statement Type:   Select
    Updatable:        No
    Deletable:        No
    Query Degree:     1
```

那么怎么解读上面的输出内容呢？主要包括如下内容：

（1）DB2 版本和生成访问计划的时间：版本为 V9.7.4，生成时间为 2013 年 2 月 28 日 11 时 35 分。

（2）数据库管理器的配置：包括并行方式（Parallelism）、CPU 速度和通信带宽。

- 并行方式可以为 None(没有并行)、Intra-Partition Parallelism(SMP 并行)和 Inter-Partition Parallelism(DPF 或者 MPP 模式)。在并行模式中，访问计划中可能出现 TQ(Table Queue)操作符，如 LTQ、BTQ、DTQ 等。TQ 以下的访问计划部分将在不同的 DB2 代理或者节点之间并行执行，而 TQ 则用于在不同的代理或者节点之间传输数据。
- CPU 速度：DB2 会自动获取服务器的 CPU 速度，对应 DBM 的参数为 CPUSPEED。一般不需要自动设置，在模拟客户环境分析 SQL 语句的访问计划时，可以手动设置。
- 通信带宽：在 DPF 环境下，各个节点相互之间通信时的带宽，对应于 DBM 参数 COMM_BANDWIDTH 的设置。

（3）数据库配置：包含内存和锁的设置，大多可以从数据库配置中找到相应的值。

- Buffer Pool Size：数据库中所有缓存池的大小总和。分析访问计划时这个参数并不可靠，因为 DB2 的表和索引都有关联的缓存池，只需关心相关的缓存池大小。
- Sort Heap Size：一个排序可用的最大内存，另外，哈希连接用的内存也是排序堆。示例中所用数据库的 SORTHEAP 配置为 AUTOMATIC（56），因而此处排序堆的大小为 56。
- Average Applications：对应数据库配置参数 AVG_APPLS，表明平均有多少用户连接这个数据库，DB2 优化器优生成访问计划时会根据这个参数来分配可用的内存资源和锁资源。

（4）语句级别上下文：包括优化级别、隔离级别和 SMP 并行度等。

- Optimization Level（优化级别）：可以为 0、1、2、3、5、7、9， DB2 根据数据库配置参数 DFT_QUERYOPT 获得默认配置，默认为 5。优化级别越高，优化器需要考虑的访问计划也就越多，编译时间也就越长。对 OLAP 应用，可以使用较高的优化级别；对于高并发的 OLTP 应用，则可以选择较低的优化级别，一般情况下，使用默认 5 就行。在当前会话中，可以为 SQL 语句设置优化级别，例如：SET CURRENT QUERY OPTIMIZATION = 3。
- Isolation Level（隔离级别）：可以为 UR、CS、RS 和 RR，请参考第 8 章相关内容。
- Query Degree：SMP 环境下处理查询的并行度，默认情况下，DB2 根据 MAX_QUERYDEGREE 和 DFT_DEGREE 计算得到。在当前会话中设置并行度的语句示例为：SET CURRENT DEGREE = '4'。

2. SQL 语句文本

原始 SQL 语句和经过优化后的 SQL 语句示例如下所示，这两条语句在语义上等价，从优化后的 SQL 语句中，可以看出编译器做了哪些重写。这个例子中，只是将 between…and 谓词转换成了等价的 <= 和>=谓词。

```
Original Statement:
------------------
select sum(ss_quantity)
from tpcds.store_sales
where ss_sold_date_sk between 2451000 and 2451500 and ss_customer_sk = 1
```

```
Optimized Statement:
-------------------
SELECT Q3.$C0
FROM
   (SELECT SUM(Q2.SS_QUANTITY)
   FROM
      (SELECT Q1.SS_QUANTITY
      FROM TPCDS.STORE_SALES AS Q1
      WHERE (Q1.SS_CUSTOMER_SK = 1) AND (Q1.SS_SOLD_DATE_SK <= 2451500) AND
              (2451000 <= Q1.SS_SOLD_DATE_SK)) AS Q2) AS Q3
```

3. 访问计划

如下所示，db2exfmt 用树型图表示访问计划，包含的节点可以分为两类：操作节点和对象节点，其中非叶节点为操作节点，叶子节点为对象节点。操作节点描述每个操作的具体信息，从上到下依次为：返回的行数、操作符名称、编号、累计的成本和 I/O 成本；对象节点包括基本表、索引、临时表和表函数等。

```
Access Plan:
-----------
     Total Cost:       22.8282
     Query Degree:         1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
                1
             GRPBY
             (   2)
             22.8277
                3
               |
             7.25133
             FETCH
             (   3)
             22.8265
                3
          /----+-----\
       7.25133        1e+07
       IXSCAN   DP-TABLE: TPCDS
       (   4)      STORE_SALES
       15.2442         Q1
          2
         |
        1e+07
 DP-INDEX: TPCDS
       SS_CUST
         Q1
```

注意：操作符的成本

访问计划图中操作符的成本是累计的，是该操作符及其子树成本的总和。因此，在分析访问计划时，应该着眼于成本增长最大的操作符。

在 DB2 中，操作符及其主要描述如表 9-1 所示。

表 9-1　DB2 访问计划中的主要操作符

操作符	描述
RETURN	在访问计划图顶部，用于返回数据
TBSCAN	表扫描，扫描基本表或者系统临时表中的每一行数据
IXSCAN	索引扫描，根据谓词搜索索引树，得到满足条件的行标示符列表（RID 列表）
FETCH	提取，根据索引扫描得到的 RID 列表，再从基本表中读取它们
FILTER	过滤器，复杂谓词（Residual Predicate）计算
SORT	排序，按某些列的值对数据行进行排序，或者对 RID 列表排序
RIDSCAN	扫描 RID 列表，从排序的 RID 列表中读取 RID，用于列表预取
IXAND	索引 AND 操作
NLJOIN	嵌套循环连接
HSJOIN	哈希连接
MSJOIN	归并连接
GROUPBY	分组与聚集
TQ	表队列，用于 DPF 或 MPP 环境下在不同的节点间交换数据，有 LTQ、BTQ、DTQ 等
SHIP	用于数据联邦
INSERT	插入操作
UPDATE	更新操作
DELETE	删除操作

访问计划图后面紧接着是每个操作符的详细信息，包括操作符的成本明细、参数和谓词。需要特别注意谓词的类型和过滤因子，请参考第 4 章 4.5.2 小节图 4-7 谓词分类与处理。在下面的示例中，该谓词为范围定界（Range-Delimiting），用于索引扫描，而过滤因子为 2.39675e-06。在 9.2.5 小节，将深入分析谓词过滤因子和基数估计的原理。

```
4) IXSCAN: (Index Scan)
        Cumulative Total Cost:        15.2442
        Cumulative CPU Cost:      217565
        Cumulative I/O Cost:      2
        Cumulative Re-Total Cost:     0.107546
        Cumulative Re-CPU Cost:   188430
        Cumulative Re-I/O Cost:   0
        Cumulative First Row Cost:    7.62028
        Estimated Bufferpool Buffers:     4

        Arguments:
        ---------
```

```
        CUR_COMM: (Currently Committed)
            TRUE
        DPESTFLG: (Number of data partitions accessed are Estimated)
            FALSE
        DPLSTPRT: (List of data partitions accessed)
            1-2
        DPNUMPRT: (Number of data partitions accessed)
            2
        GLOBLOCK: (Global Lock intent)
            INTENT SHARE
        LCKAVOID: (Lock Avoidance)
            TRUE
        MAXPAGES: (Maximum pages for prefetch)
            2
        PREFETCH: (Type of Prefetch)
            NONE
        ROWLOCK : (Row Lock intent)
            SHARE (CS/RS)
        SCANDIR : (Scan Direction)
            FORWARD
        SKIP_INS: (Skip Inserted Rows)
            TRUE
        TABLOCK : (Table Lock intent)
            INTENT SHARE
        TBISOLVL: (Table access Isolation Level)
            CURSOR STABILITY

        Predicates:
        ----------
        2) Start Key Predicate,
            Comparison Operator:      Equal (=)
            Subquery Input Required:  No
            Filter Factor:            2.39675e-06

            Predicate Text:
            --------------
            (Q1.SS_CUSTOMER_SK = 1)



        2) Stop Key Predicate,
            Comparison Operator:      Equal (=)
            Subquery Input Required:  No
            Filter Factor:            2.39675e-06

            Predicate Text:
            --------------
            (Q1.SS_CUSTOMER_SK = 1)
... ...
```

4. 访问计划中用到的表和索引等数据库对象

该访问计划用到的数据库对象为表 TPCDS.STORE_SALES 和索引 SS_CUST，如下所示，db2exfmt 打印出的信息还包含它们的基本信息和统计信息。

```
Objects Used in Access Plan:
---------------------------

        Schema: TPCDS
        Name:   STORE_SALES
        Type:   Data Partitioned Table
                        Time of creation:            2012-07-28-09.06.14.407324
                        Last statistics update:       2013-04-16-00.00.00.000000
                        Number of columns:            23
                        Number of rows:              10000000
                        Width of rows:               49
                        Number of buffer pool pages:   35008
                        Number of data partitions:     8
                        Distinct row values:          No
                        Tablespace name:             <VARIOUS>
                        Tablespace overhead:          7.500000
                        Tablespace transfer rate:      0.060000
                        Source for statistics:        Single Node
                        Prefetch page count:          32
                        Container extent page count:   32
                        Table overflow record count:   0
                        Table Active Blocks:          -1
                        Average Row Compression Ratio:  2.18915
                        Percentage Rows Compressed:    98.8676
                        Average Compressed Row Size:   56

        Schema: TPCDS
        Name:   SS_CUST
        Type:   Partitioned Index
                        Time of creation:            2012-10-13-23.44.40.234940
                        Last statistics update:       2013-04-16-00.00.00.000000
                        Number of columns:            1
                        Number of rows:              10000000
                        Width of rows:               -1
                        Number of buffer pool pages:   35008
                        Distinct row values:          No
                        Tablespace name:             <VARIOUS>
                        Tablespace overhead:          7.500000
                        Tablespace transfer rate:      0.060000
                        Source for statistics:        Single Node
                        Prefetch page count:          32
                        Container extent page count:   32
                        Index clustering statistic:    93.000000
                        Index leaf pages:             432
                        Index tree levels:            2
                        Index full key cardinality:    413696
                        Index first key cardinality:   413696
                        Index first 2 keys cardinality: -1
                        Index first 3 keys cardinality: -1
                        Index first 4 keys cardinality: -1
                        Index sequential pages:        269
                        Index page density:           59
                        Index avg sequential pages:    0
```

```
Index avg gap between sequences:0
Index avg random pages:        0
Fetch avg sequential pages:     -1
Fetch avg gap between sequences:-1
Fetch avg random pages:        -1
Index RID count:               1425394
Index deleted RID count:        0
Index empty leaf pages:         0
Base Table Schema:              TPCDS
Base Table Name:                STORE_SALES
Columns in index:
      SS_CUSTOMER_SK(A)
```

9.2.3 表扫描与索引扫描

当优化器生成访问计划时，有两个最基本的问题：用什么样的方式访问表中的数据；用什么样的方式和顺序处理表之间的连接。本小节将介绍访问表的方式，9.2.4 节介绍表连接方式。

访问表的方式称为“访问路径”（Access Path）。访问路径分索引扫描和表扫描。一般来讲，索引扫描比表扫描快，但也并不绝对。具体来说，有如下几种：

- 表扫描：TBSCAN；
- 索引扫描：IXSCAN + FETCH；
- 索引列表预取：IXSCAN + List Prefetch；
- 索引与操作：IXSCAN + IXAND，能利用多个索引并进行“与操作”；
- 索引或操作：IXSCAN + SORT +RIDSCAN，能利用多个索引并进行“或操作”。

注意：索引或操作没有显式的 IXOR 操作符

索引“或操作”没有显式的 IXOR 操作符，实际上是将索引扫描后的各个 RID 列表排序，然后用 RIDSCAN 操作对这些有序的 RID 列表进行归并扫描，从而实现索引“或操作”。

在本书第 4 章 4.5 节中已经讲过表扫描和索引扫描，这一节将重点讲解索引列表预取、索引“与操作”和索引“或操作”。

1. 索引列表预取

索引扫描有一种方式称为 List Prefetch，先通过索引扫描得到满足谓词条件的所有 RID，然后将这些 RID 进行排序，再用它们访问数据页。这样可以对相邻的数据行进行连续的访问，从而通过预取减少 I/O 次数。与随机 I/O 相比，这种方式的开销要小得多，因为如果缓冲池没法放下所有的页，随机访问可能导致一个页面从磁盘中读入多次。

下面是 List Prefetch 的例子：访问计划中的 SORT 操作用于将索引扫描得出的 RID 进行排序，然后再用预取的方式读取基本表的数据。预取能够减少磁盘 I/O 次数，从而提高数据访问的性能。

```
               455.385
                FETCH
                (   9)
                308.619
                61.2878
              /----+---\
         455.385       15009
         RIDSCN    TABLE: TPCD
         (  10)   L_SUMMARY2
         219.093
         17.4697
           |
           |
         455.385
         SORT
         (  11)
         219.091
         17.4697
           |
         455.385
         IXSCAN
         (  12)
         218.559
         17.4697
           |
          15009
      INDEX: TPCD
     L_SUMMARY2_IDX
```

2. 索引与操作

当 SQL 语句在同一个表上有多个谓词，并可以使用多个索引扫描时，DB2 使用 IXAND 操作符来执行索引与（Index Anding）操作。索引与操作将多个索引得到的 RID 列表合并，只取共同存在的 RID。

例如，LINEITEM 表上有两个索引：orderkey 列上的 L_OK 和 prod_key 列上的 L_PD。下面这条查询在 order_key 上的谓词可以使用索引 L_OK，在 prod_key 上的谓词可以使用索引 L_PD。

```
SELECT   sum(quantity)
FROM     TPCD.LINEITEM
WHERE    order_key in (103982, 1039473, 1039294)
AND      prod_key = 498239;
```

相应的访问计划如下所示，IXAND 节点将两个 IXSCAN 连接起来，而在 IXAND 上采取了 LIST PREFETCH。

```
                     |
                  4.4314
                   FETCH
                   (   3)
                   5475.6
```

```
                  1952.4
                /----+---\
            886.281    1.50002e+08
            RIDSCN      TABLE: TPCD
            (   4)      LINEITEM
            4027.9
            1100.96
               |
             886.281
              SORT
             (   5)
             4027.9
             1100.96
                |
              886.281
              IXAND
              (   6)
              4026.01
              1100.96
          /-----+-----\
      248752          534445
      IXSCAN          IXSCAN
      (   7)         (   8)
      1480.95         2509.07
      430.024         670.935
        |               |
     1.50002e+08     1.50002e+08
   INDEX: TPCD     INDEX: TPCD
      L_OK            L_PD
```

3. 索引或操作

如果 SQL 语句中的多个谓词是“或（OR）”组合在一起，并同时可以使用多个索引。这时，DB2 通过 Index Oring 操作，将多个索引扫描得到的 RID 进行合并，并将重复的 RID 去掉。与 Index Anding 操作不同，索引“或操作”在访问计划中没有显式的操作符（比如 IXOR）。

使用索引“或”的访问计划示例如下，它同时使用了三个索引扫描，并且对每个索引扫描得到的 RID 进行排序，然后对三个 RID 列表上直接做 RIDSCAN 进行 RID 的合并操作，最后从基本表上提取数据。

```
                            |
                         59537.2
                          FETCH
                          (   3)
                          62819.1
                           37361.7
                         /----+---\
                    59537.2    1.50002e+08
                     RIDSCN    TABLE: TPCD
                      (   4)      LINEITEM
                     803.781
                     82.2908
```

```
                      |
                      |
      +-----------------+-----------------+
    59383.3          150.002           4.00006
    SORT             SORT              SORT
    (   5)           (   7)            (   9)
    653.127          75.5534           75.1057
    76.2908             3                 3
      |                |                 |
    59383.3          150.002           4.00006
    IXSCAN           IXSCAN            IXSCAN
    (   6)           (   8)            (  10)
    510.773          75.4177           75.1022
    76.2908             3                 3
      |                |                 |
  1.50002e+08      1.50002e+08       1.50002e+08
    INDEX:           INDEX:            INDEX:
   TPCD.L_SD       TPCD.L_SK_PK       TPCD.L_OK
```

9.2.4 嵌套循环连接、归并连接和哈希连接

如果需要连接两个以上的表，优化器每次也最多只能连接两个表，并保存中间结果，再与其他表做两两连接。优化器在考虑两个表的连接操作时，主要考虑三种连接方法：嵌套循环连接、归并连接和哈希连接。本节主要讨论这三种连接方法。

注意：星型连接

对于用星型模型建模的 OLAP 应用，DB2 可为复杂查询生成星型连接的访问计划，这样可以一次性连接多个表。DB2 中支持三种星型连接：

- Cartesian Hub：维度表之间先做笛卡尔连接，然后再和事实表做连接。
- Dynamic Index Anding：每个维度表通过事实表上的索引与事实表进行半连接操作，得到事实表中满足各连接谓词的 RID 列表，然后用动态位图进行索引与操作，然后从事实表中提取数据，最后再与维度表做连接。
- Zigzag Join：DB2 V10 提供的新技术，维度表通过事实表中的多列索引一次性完成多表的星型连接操作。

注意：外表和内表

对于两两连接方法，连接的两个表一个为外表（outer table）；另一个为内表（inner table）。外表也称为驱动表，处理连接时，对于外表的每一行，都会去内表中查找是否有相应的匹配行。

优化器根据所选连接方法的类型和成本决定哪个是外表、哪个是内表。

这三种连接方法的访问计划示意图如图 9-4 所示，各有特点：

- 嵌套循环连接：两层循环处理连接，适用于内表较小或者内表上可以使用索引扫描的场景。
- 归并连接：内表和外表都需要有序，这可以用显式的 SORT 来保证，也可以是索引中

本来就是有序的数据。归并连接的谓词只能是等于谓词。

- 哈希连接：内表一般较小，用内表构建哈希表。对于外表的每一行，在这个哈希表中进行快速查找。哈希连接的谓词也只能是等于谓词。

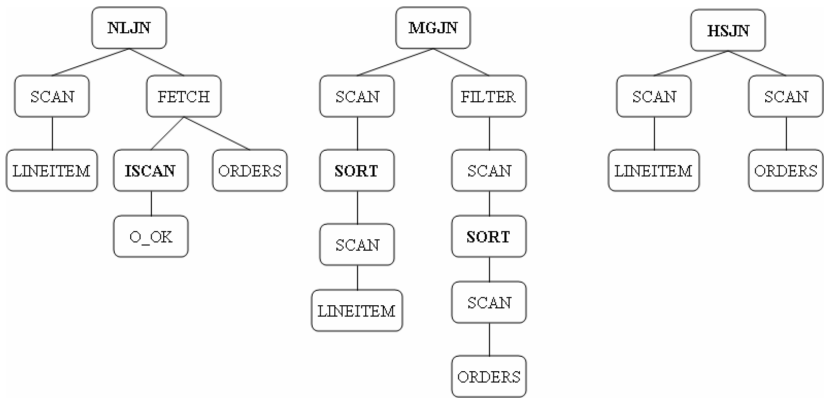


图 9-4　三种连接的访问计划示意图

1. 嵌套循环连接

循环嵌套连接的运行方式如图 9-5 所示。外表只扫描一次，而针对外表中的每一行，都要扫描一次内表，查找是否有匹配的行。

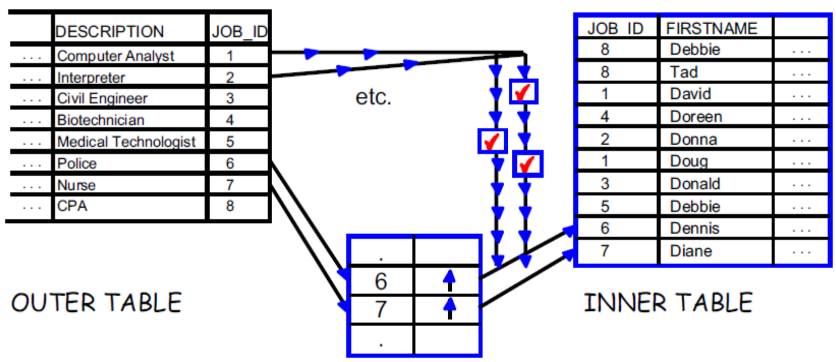


图 9-5　嵌套连接

嵌套循环连接根据外表的当前行，查找内表有两种方法：

（1）对内表全扫描：即读取内表中的每一行，与外表中的当前行进行匹配，这适合于内表较小的情形。

（2）对内表索引扫描：当连接的谓词可以使用内表的索引时，外表的每一个值都能在内表的索引上找到。这也是嵌套循环连接最常见的形式。

下面是嵌套循环连接的例子，其中 TELLER 是外表，HISTORY 是内表。从访问计划中可以看出，内表 HISTORY 使用了索引扫描，对于外表 TELLER 的每一行，都会对内表进行一次索引扫描。

```
SELECT HISTORY.BRANCH_ID, TELLER.TELLER_NAME,
 HISTORY.ACCTNAME,
   HISTORY.ACCT_ID, HISTORY.BALANCE
```

```
    FROM HISTORY AS HISTORY, TELLER AS TELLER
    WHERE HISTORY.TELLER_ID = TELLER.TELLER_ID AND HISTORY.BRANCH_ID = 25
    ORDER BY HISTORY.BRANCH_ID ASC, HISTORY.ACCT_ID ASC ;

                       794.52
                       TBSCAN
                       (   2)
                       3827.09
                       317.376
                         |
                       794.52
                       SORT
                       (   3)
                       3827.02
                       317.376
                         |
                       794.52
                       NLJOIN
                       (   4)
                       3826.48
                       317.376
                /----------+---------\
            1000                    0.79452
           FETCH                    FETCH
           (   5)                   (   7)
           159.046                  23.0878
             33                    1.79452
         /---+---\                /---+---\
      1000        1000       0.79452      79452
     IXSCAN   TABLE: ADMIN     IXSCAN   TABLE: ADMIN
     (   6)       TELLER       (   8)      HISTORY
     52.0664                  12.8738
        4                        1
       |                        |
      1000                     79452
  INDEX: ADMIN              INDEX: ADMIN
     TELLINDX                  HISTIX1
```

注意：嵌套循环连接当仁不让的情形

对于不等谓词（即>、<、>=、<= 和 !=）的连接，只能使用嵌套循环连接。对于两个表的笛卡尔积（即没有连接谓词），也只能使用嵌套循环连接，如 SELECT * FROM T1, T2。

2. 归并连接

归并连接的谓词必须是等于谓词（即具有 table1.column = table2.colum 形式），并且还要求根据连接列对两个表进行排序。

归并连接的运行方式如图 9-6 所示。由于外表和内表都是有序的，所以外表和内表都只扫描一次。

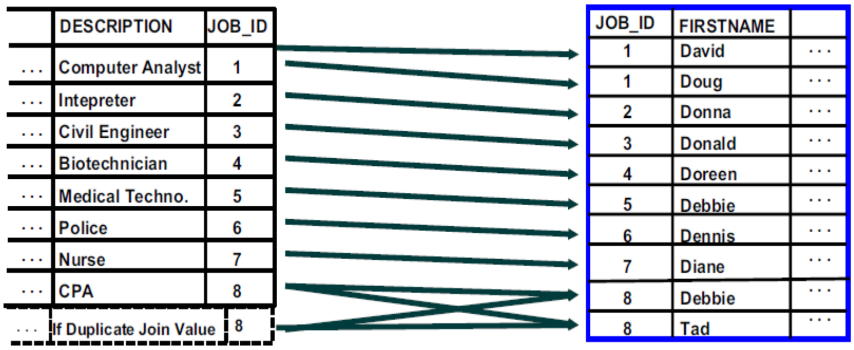


图 9-6　归并连接

归并连接的访问计划示例如下所示，和嵌套循环连接一样，TELLER 是外表，HISTORY 是内表。可以发现外表 TELLER 通过 IXSCAN（索引数据本来就是有序的）来保证连接字段有序；内表 HISTORY 有个显式 SORT 操作从而保证连接字段有序。

```
    SELECT HISTORY.BRANCH_ID, TELLER.TELLER_NAME,
     HISTORY.ACCTNAME,
       HISTORY.ACCT_ID, HISTORY.BALANCE
       FROM HISTORY AS HISTORY, TELLER AS TELLER
       WHERE HISTORY.TELLER_ID = TELLER.TELLER_ID AND HISTORY.BRANCH_ID = 25
       ORDER BY HISTORY.BRANCH_ID ASC, HISTORY.ACCT_ID ASC ;

                        794.52
                        MSJOIN
                        (   4)
                        830.571
                          375
                 /-------+------\
            1000                0.79452
           FETCH                FILTER
           (   5)               (   7)
           159.046               671.23
             33                   342
         /---+---\                 |
       1000        1000        794.52
      IXSCAN    TABLE: ADMIN     TBSCAN
      (   6)       TELLER       (   8)
      52.0664                   671.23
         4                       342
        |                         |
       1000                    794.52
     INDEX: ADMIN               SORT
        TELLINDX               (   9)
                               671.229
                                342
                                 |
                               794.52
                               TBSCAN
                               (  10)
```

```
                    670.873
                      342
                       |
                     79452
                  TABLE: ADMIN
                    HISTORY
```

3. 哈希连接

哈希连接只能处理一个或多个等于连接谓词，并要求连接谓词两端的字段类型相同，且不能含有表达式。就 CHAR 类型而言，长度必须相同；就 DECIMAL 类型而言，精度和小数位必须相同。

注意：DB2 V10 对哈希连接的增强

DB2 V10 增强了哈希连接的能力，能将含有表达式或者字段类型不匹配的连接谓词作为哈希连接的连接谓词，比如 T1.C1 = T2.C2+1，或者 T1.C1_INT = T2.C2_DECIMAL 等。不过，从性能上考虑，应该避免使用复杂表达式或者类型不匹配的连接谓词。

哈希连接的运行方式如图 9-7 所示，分为两个阶段：

（1）构建哈希表（build table）：首先扫描内表，根据连接谓词的字段计算哈希键（hash key），构建一个哈希表（这个阶段称为构建表，build table）。哈希表被分为几个分区，如果内存中没有足够的空间容纳整个表，则有些分区将被写入磁盘上的临时表。

（2）探测表（probe table）：扫描外表，对于外表的每一行，对连接列应用同一哈希算法计算哈希键，然后根据哈希键在哈希表中查找内表的匹配记录。如果哈希键对应的哈希表分区在内存中，则立即返回匹配连接的结果。如果该分区被写入到磁盘上的临时表中，则外表的行也被写入临时表。

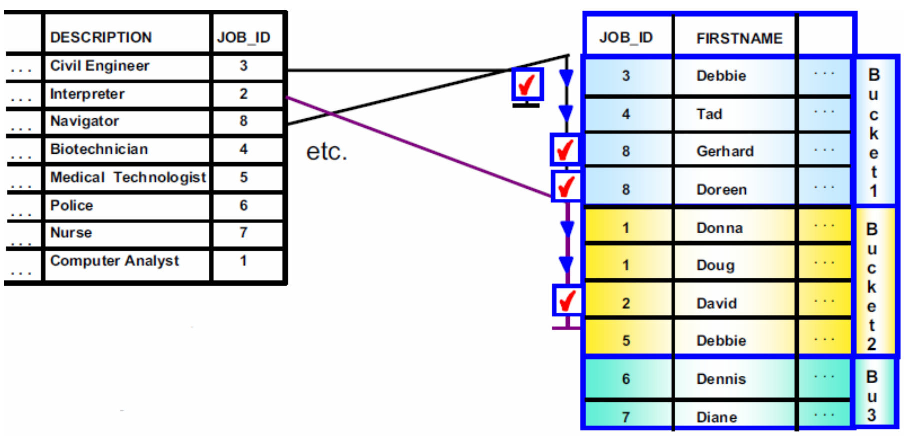


图 9-7 哈希连接

注意：哈希连接使用的内存池为 SORTHEAP

哈希连接构建哈希表时使用的内存池为 SORTHEAP，如果 SORTHEAP 的内存不够，则会溢出到磁盘上，导致磁盘 I/O 溢出。因此，为了提高哈希连接的性能，避免 I/O 溢出，应该适当调整 SORTHEAP 的大小。

不同于嵌套循环连接和归并连接，HISTORY 是外表，TELLER 是内表，用于构建哈希表。哈希连接的访问计划示例如下所示：

```
SELECT HISTORY.BRANCH_ID, TELLER.TELLER_NAME,
 HISTORY.ACCTNAME,
   HISTORY.ACCT_ID, HISTORY.BALANCE
   FROM HISTORY AS HISTORY, TELLER AS TELLER
   WHERE HISTORY.TELLER_ID = TELLER.TELLER_ID AND HISTORY.BRANCH_ID = 25
   ORDER BY HISTORY.BRANCH_ID ASC, HISTORY.ACCT_ID ASC ;

                       794.52
                       HSJOIN
                       (   4)
                       722.496
                       336.17
               /----------+---------\
          794.52                  1000
          FETCH                 FETCH
          (   5)                (   9)
          563.302                159.046
          303.17                  33
         /---+---\              /---+---\
    794.52       79452      1000        1000
    RIDSCN   TABLE: ADMIN    IXSCAN   TABLE: ADMIN
    (   6)      HISTORY     (  10)      TELLER
    13.6858                52.0664
      1                      4
      |                     |
    794.52                  1000
    SORT               INDEX: ADMIN
    (   7)                TELLINDX
    13.6853
      1
      |
    794.52
    IXSCAN
    (   8)
    13.3792
      1
      |
     79452
 INDEX: ADMIN
    HISTIX1
```

9.2.5　基数估计和统计信息

访问计划中每个操作符上有 3 个关键的数字：基数、成本和 I/O 成本。在 9.1 节中，我们已经了解优化器基于成本来评估候选的访问计划。基数估计是成本评估的关键，而优化器依赖于准确的统计信息来进行基数估计。本节将深入讨论这个问题。

优化器在基数估计和成本评估的过程中，有 3 个度量标准：

- Cost（成本）：成本表现为 CPU、I/O 和通信等所需要的开销，成本的单位为 timerons，

这是一个人造的单位，不能解释为实际时间单位。

- Cardinality（基数）：返回数据集的行数。对于访问计划中的每个操作符，优化器根据输入数据集的行数和操作符上的谓词（或者聚集操作）估计输出数据集的行数。
- Filter Factor（过滤因子，简记为 ff）：表示数据集满足谓词条件的比率，它的值介于 0~1 之间，0 表示没有数据行满足谓词条件。在访问计划操作符的详细信息中，每个谓词都有一个 Filter Factor。

> **注意：过滤因子 vs 选择度**
>
> 过滤因子的另一种说法为选择度（selectivity），谓词的选择度高对应过滤因子的值较小，谓词的选择度低对应过滤因子的值较大。

优化器正是凭借基数估计（Cardinality Estimation）来计算每个操作符的成本以及整个查询的成本。要想基数估计精确，就需要有准确的统计信息，从这个角度上说，统计信息是优化器的“导航仪”。

DB2 中的统计信息主要包括以下内容：

- 基本统计信息：包括表、列和索引的统计信息。
- 分布统计信息：频率（frequency）统计信息和分位数（quantile）统计信息。
- 列组统计信息：计算具有相关性的列组（column group）统计信息，为联合谓词的过滤因子提供更精确的估计。

本节将介绍各种统计信息，并讨论优化器如何利用这些统计信息进行基数估计。

1. 如何查看现有的统计信息

首先讲解如何查看现有统计信息。

（1）直接查询 syscat 或 sysstat 模式下的视图

相关视图如表 9-2 所示。包括 syscat 视图和 sysstat 视图。实际上，sysstat 视图正是基于相应的 syscat 视图而定义的。

表 9-2　统计信息视图

统计信息	syscat 视图	sysstat 视图
表的统计信息	TABLES	TABLES
列的基本统计信息	COLUMNS	COLUMNS
索引的统计信息	INDEXES	INDEXES
列的分布统计信息	COLDIST	COLDIST
列组统计信息	COLGROUPS	COLGROUPS

（2）解释 SQL 语句时使用 snapshot 选项

在解释 SQL 语句时使用 snapshot 选项，这样 DB2 将编译 SQL 语句时使用的统计信息也包含在访问计划的输出中，这样大大方便了对访问计划的分析。

```
db2 explain plan with snapshot for select ...;
db2exfmt  -d <db_name> -1 -o <outputfile>
```

```
或者
db2 set current explain mode explain
db2 set current explain snapshot explain
select ...
 db2 set current explain mode no
 db2 set current explain snapshot no
db2exfmt -d <db_name> -1 -o <outputfile>
```

（3）使用 db2cat 工具打印出相关表的统计信息

使用 db2cat 的命令示例如下，其中，-d 选项指定数据库，-s 选项指定模式，-n 选项指定表名，-o 选项指定输出文件：

```
db2cat -d tpcds -s TPCDS -n store_sales -o store_sales.cat
```

（4）使用 db2look 工具导出相关表的统计信息

使用 db2look 的命令示例如下，其中，-d 选项指定数据库，-z 选项指定模式，-t 选项指定表名，-m 表示导出统计信息，-o 选项指定输出文件。db2look 的输出中用 update 语句的形式显示了相应的统计信息。

```
db2look -d tpcds -z TPCDS -t store_sales -m -o store_sales.cat
```

2. 基本统计信息与基数估计

接下来，介绍基本统计信息以及优化器如何使用这些统计信息进行基数估计。

（1）表的基本统计信息

表的基本统计信息主要有：

- CARD：表中的行数（基数）。
- FPAGES：当前使用的页面数。
- NPAGES：包含记录行的页面数。
- OVERFLOW：溢出的行数。
- AVGROWSIZE：行的平均长度。
- ACTIVE_BLOCKS：表示 MDC 包含数据的块数。

表的统计信息可以从视图 syscat.tables 或者 sysstat.tables 中查到。如下所示，表 TPCDS.STORE_SALES 的行数为 10000000，占用存储为 35008 页（FPAGES），实际存储数据行的页 NPAGES 为 34976，行的平均长度 AVGROWSIZE 为 56 字节。

```
    select  CARD,  NPAGES,  FPAGES,  AVGROWSIZE,OVERFLOW  from  sysstat.tables  where
TABSCHEMA='TPCDS' and TABNAME='STORE_SALES'

    CARD          NPAGES          FPAGES            AVGROWSIZE OVERFLOW
    ------------ -------------- ---------------- ---------- ---------
       10000000           34976            35008         56         0
```

注意：什么时候 REORG？

当表统计信息中的 FPAGES 与 NPAGES 相差加大，或者 OVERFLOW 的值较大时，说明表的存储碎片较多，就应该运行 REORGCHK 或 REORG 重组表，然后 RUNSTATS 更新统计信息。

（2）列的基本统计信息

列的基本统计信息主要有：

- COLCARD：列上不同值的数目。
- NUMNULLS：列上空值数据。
- HIGH2KEY：列上的第二大值。
- LOW2KEY：列上的第二小值。

列的统计信息可以从视图 syscat.columns 或者 sysstat.columns 中查到。如下所示，表 STORE_SALES 中字段 SS_EXT_SALES_PRICE 上不同值（COLCARD）的数目为 292，空值数据为 0，而第二大值（HIGH2KEY）为 990.00，第二小值（LOW2KEY）为 4.00。

```
    select   COLCARD,  NUMNULLS,  HIGH2KEY,  LOW2KEY  from  sysstat.columns  where
TABSCHEMA='TPCDS' and TABNAME='STORE_SALES' and COLNAME='SS_EXT_SALES_PRICE'

    COLCARD      NUMNULLS     HIGH2KEY      LOW2KEY
    ------------ ------------ ------------ ------------
             292            0    +00990.00     +00004.00
```

注意：为什么使用 HIGH2KEY 和 LOW2KEY？

为什么 DB2 不统计列上的最大最小值，退而求其次，统计第二大和第二小的值呢？因为这样可以避免一些异常数据带来的误差。实践证明，使用 HIGH2KEY 和 LOW2KEY 来估计基数效果更好。

（3）索引的基本统计信息

索引的基本统计信息主要有：

- NLEAF：叶子节点的页面数。
- NLEVELS：索引的层级。
- INDCARD：索引项的数目，这个数目与表的基数可能不同，因为索引项与数据行并不是一一对应的关系。
- FIRSTKEYCARD：组合索引上第一个列的不同值数目。
- FIRST2KEYCARD/FIRST3KEYCARD/FIRST4KEYCARD：组合索引中前 2、3、4 个列的基数，当这些项不适用于该索引时（比如单键索引），值为－1。
- FULLKEYCARD：索引全部列的不同值的数目。

索引的统计信息可以从视图 syscat.indexes 或者 sysstat.indexes 中查到。如下所示，索引 C_SALESDATE 是定义在表 STORE_SALES 上的字段 SS_SOLD_DATE_SK 的单键索引，叶子页面数为 267，而索引的层级为 2，共有 3651 个不同的索引键值，索引项为 1425394。

```
    select   NLEAF,  NLEVELS,  FIRSTKEYCARD,  INDCARD  from  sysstat.indexes  where
TABSCHEMA='TPCDS' and TABNAME='STORE_SALES' and INDNAME='SS_SOLDDATE'

    NLEAF                NLEVELS FIRSTKEYCARD         INDCARD
    ----------------- ------- -------------------- ---------------------
                  267       2                 1824                1425394
```

> **注意：索引的其他统计信息**
>
> 索引上还有其他的统计信息，比如 SEQUENTIAL_PAGES、DENSITY、CLUSTERRATIO、CLUSTERFACTOR 等，这些统计信息对于估计索引扫描所需 I/O 成本时非常重要，这些值不理想时也能作为是否需要 REORG 的依据。

（4）基于基本统计信息的基数估计

只有基本统计信息的情况下，DB2 优化器认为所有的数据都是均匀分布的。优化器对下面三类谓词的过滤因子计算公式如下所示：

- 局部等于谓词（如 C1=7）：

```
ff = 1/colcard（C1）
```

- 范围谓词（如 key1 < C1 <= key2）：

```
ff=（key2-key1）/（high2key-low2key）
```

- 连接谓词（如 C1=C2）：

```
ff=1/MAX（colcard（C1），colcard（C2））
```

对于单个表 T 上的谓词条件，相应的基数计算公式为：

```
card(T) * ff
```

对于两个表 T1 和 T2 上的连接谓词，相应的基数计算公式为：

```
card(T1) * card(T2) * ff
```

例如，对于查询 select count(*)from TPCDS.STORE_SALES where SS_EXT_SALES_PRICE = 50.00，根据表的基本统计信息，得到 card(STORE_SALES)=10000000，而根据列的统计信息，得到 colcard（SS_EXT_SALES_PRICE）=292，于是谓词 SS_EXT_SALES_PRICE = 50.00 的过滤因子 ff=1/colcard（SS_EXT_SALES_PRICE）=1/292 ≈ 0.00315426，于是基数估计为 card（STORE_SALES）×ff ≈ 10000000 ×0.00315426 = 31542.6。

而对于范围谓词，例如 SELECT count（*）FROM TPCDS.STORE_SALES WHERE SS_EXT_SALES_PRICE > 50.00 and SS_EXT_SALES_PRICE <=150.00，谓词的过滤因子按照公式为 ff=（key2-key1）/（high2key-low2key）=（150.00-50.00）/（990.00-4.00）≈0.10141988，其中 990.00 和 4.00 分别为列 SS_EXT_SALES_PRICE 的 high2key 和 low2key。于是基数估计为 card×ff≈10000000×0.10141988=1014129.88。

> **注意：基本统计信息的收集**
>
> 在表或者索引上运行 RUNSTATS 命令就可以收集基本的统计信息，比如，RUNSTATS ON TABLE TPCDS.STORE_SALES AND INDEXES ALL。详情请参考本书第 7 章 7.2 节。

3. 分布统计信息

如前所述，在只有基本统计信息的情况下，DB2 优化器认为所有的数据都是均匀分布的。

但这与实际的情况可能不一致，比如，列上某个值的数目比较多或者某区间上的数据比较多，使用均匀分布进行基数估计就不准确了。

例如，尽管 SS_EXT_SALES_PRICE 的值范围（销售价格区间）为 4.00～990.00，但是大部分的销售价格小于 200.00。如前面的分析，对于下面的查询，优化器利用基本统计信息估计的基数为 1014129.88，实际的查询结果则为 3618913，如下所示，存在较大偏差：

```
    SELECT  count(*)  FROM  TPCDS.STORE_SALES  WHERE  SS_EXT_SALES_PRICE  >  50.00  and
SS_EXT_SALES_PRICE <=150.00
    1
    -----------
       3618913

      1 record(s) selected.
```

因此，当表中存储了很多重复值，或者数据在表中并非均匀分布的时候，就需要利用分布统计信息对基数估计进行修正。

DB2 提供了两种分布统计信息：频率（frequency）统计信息和分位数（quantile）统计信息。

注意：分布统计信息的收集

可以使用 WITH DISTRIBUTION 选项执行 RUNSTATS 命令，来收集分布统计信息，例如，RUNSTATS ON TABLE TPCDS.STORE_SALES WITH DISTRIBUTION，收集所有列的分布统计信息。

收集分布统计信息要比收集基本统计信息开销更大，对于值分布均匀的列，不必收集分布统计信息。RUNSTATS 命令能指定只对某些列收集分布统计信息。

另外，RUNSTATS 命令也能使用 NUM_FREQVALUES(频率)和 NUM_QUANTILES（分位数）选项来定制分布统计信息的数量，默认情况下，DB2 收集 10 个频率值，20 个分位数。

（1）频率统计信息

频率统计信息记录出现频率最高的几个值以及出现次数。例如，EMPLOYEE 表上列 JOB 的频率统计信息如下所示：

```
    JOB      COUNT(JOB)
    -------- -----------
    DESIGNER          10
    CLERK              8
    MANAGER            7
    OPERATOR           6
    FIELDREP           5
    ANALYST            3
    SALESREP           2
    PRES               1
```

对于查询 select count（*）from EMPLOYEE where job = ‘DESIGNER’，如果使用基本统计信息，表的基数为 42，列 Job 的基数为 8，则基数估计为 card × 1 / colcard(job) = 42/8=5.25。如果使用频率统计信息，则能精确地得到基数为 10。

（2）分位数统计信息

分位数统计信息实际上就是统计数据的区间分布，即柱状图。例如，列 C1 包含下面的值，其数据分布区间并不是均匀的：

```
  0.0
  5.1
  6.3
  7.1
  8.2
  8.4
  8.5
  9.1
 93.6
100.0
```

分位数统计信息如下所示，数据按从小到大排序后，处于第 1 个位置的值为 0.0，第 4 个位置的值为 7.1 等。

```
K    K-quantile
--   ---------
1    0.0
4         7.1
7         8.5
10     100.0
```

用基本统计信息和分位数统计信息分别计算下面两个谓词的基数：

● C1 <= 8.5：表上共有 7 条记录满足这个谓词。

　　使用基本统计信息，基数估计为（KEY2－KEY1）/（HIGH2KEY－LOW2KEY）×Card =（8.5－5.1）/（93.6－5.1）×10 = 0.38，存在很大的偏差。而使用分位数统计信息，由于第 7 个位置的值为 8.5，直接得到基数为 7，与实际情况完全一致。

● C1 <= 10：表上共有 8 条记录满足这个谓词。

　　使用基本统计信息估计的基数为（10－5.1）/（93.6－5.1）×10 =0.55。而使用分位数统计信息，需要将这个谓词作为两部分来进行估算：C1 <= 8.5 的基数为 7，而 C1> 8.5 AND C1 <= 10 的基数估计为（10－8.5）/（100－8.5）×（10－7）=0.05。因此，谓词 C1 <= 10 估计为 7.05，比使用基本统计信息要精确得多。

我们再用统计分布信息来分析这个查询的基数估计：

```
    SELECT count（*） FROM TPCDS.STORE_SALES WHERE SS_EXT_SALES_PRICE > 50.00 and
SS_EXT_SALES_PRICE <=150.00
```

对表 TPCDS.STORE_SALES 收集分布统计信息后，使用 EXPLAIN PLAN WIHT SNAPSHOT 选项打印出其访问计划。

如下所示，在访问计划用到的对象信息中，可以找到列 SS_EXT_SALES_PRICE 的分布统计信息，DB2 收集了 10 个频度统计信息和 20 个分位数统计信息。在实际过程中，优化器将同时使用频度统计信息和分位数统计信息进行基数估计。这里为简化问题，在估计这个区间谓词的过滤因子时忽略掉频度统计信息的影响，仅考虑用分位数统计信息进行估计。从分位统计信

息中可以看到，第 1620000 个位置的值为 48.00，而第 5305001 个位置的值为 148.00，因此粗略估计，大于 50.00 而小于等于 150 的行数目约为 5305001－1620000=3685001，而如前面所示，该查询的实际结果为 3618913，非常接近。

```
Column Information:
--------------------
        Number:                                       16
        Name:                                         SS_EXT_SALES_PRICE
        Statistics Available:                            Yes

        Column Statistics:
        ------------------
                Schema name of the column type:                SYSIBM
                Name of column type:                          DECIMAL
                Maximum column length:                        4
                Scale for decimal or timestamp column:          2
                Number of distinct column values:              292
                Average column length:                        5
                Number of most frequent values:                 10
                Number of quantiles:                          20
                Second highest data value:                     +00990.00
                Second lowest data value:                      +00004.00
                Column sequence in partition key:               0
                Average number of sub-elements:                 -1
                Average length of delimiters:                   -1

        Column Distribution Statistics:
        -------------------------------
                Frequency Statistics:
                Valcount        Value
                --------------------------------
                140000         +00192.00
                115000         +00076.00
                115000         +00040.00
                110000         +00024.00
                110000         +00100.00
                105000         +00156.00
                105000         +00064.00
                105000         +00090.00
                100000         +00102.00
                100000         +00016.00

                Quantile Statistics:
                Valcount        Value          Distcount
                ----------------------------------------------
                40000          +00002.00              0
                565000         +00018.00              0
                1085000               +00034.00             0
                1620000               +00048.00             0
                2105000               +00062.00             0
                2675000               +00076.00             0
                3225000               +00090.00             0
```

```
        3750000             +00104.00         0
        4255001             +00118.00         0
        4770001             +00134.00         0
        5305001             +00148.00         0
        5855001             +00162.00         0
        6330001             +00176.00         0
        6940001             +00192.00         0
        7375001             +00224.00         0
        7895001             +00300.00         0
        8420001             +00384.00         0
        8960001             +00504.00         0
        9475001             +00670.00         0
        10000000            +00990.00         0
```

因此，如果一个数据库包含很多重复的值，或者数据分布不均匀，那么可以收集分布统计信息，这样 DB2 优化器就能更精确地进行基数估计，从而为 SQL 语句生成更好的访问计划。

注意：含参数和主变量的谓词

如果 SQL 语句中的谓词使用的是参数或者主变量，比如 C1 = ?，则优化器用不上分布统计信息，只能使用基本统计信息来估计基数。

4. 列组统计信息

如果 SQL 语句中有两个或者多个谓词，优化器在估计基数时假定这两个谓词是独立的。但是，在某些情况下，这两个谓词或者多个谓词并不独立，也就是说这些列上的数据具有相关性，这时，优化器估计的基数就会存在偏差。

例如，EMPLOYEE 表上共有 42 条记录，满足谓词 JOB=‘ANALYST’的数据行数为 3，而满足 WOKRDEPT=‘C01’的行数为 4。在收集了分布统计信息后，优化器能精确地估计这两个谓词的过滤因子，分别为 3/42= 0.0714286 和 4/42= 0.0952381。

对于查询 select * from employee where job =‘ANALYST’and workdept =‘C01’，优化器会假定这两个谓词是独立的，于是估计基数为 Card×（3/42）×（4/42）= 0.28571。其访问计划如下所示：

```
      0.28571
      TBSCAN
      (   2)
      7.65299
        1
        |
        42
  TABLE: DB2INST1
     EMPLOYEE
        Q1
```

但是，这个查询实际返回行数为 3。这里的基数估计偏差正是由于 workdept 和 job 的数据具有相关性导致的。这时可以在 workdept 和 job 上收集列组统计信息，修正优化器的基数估计。

收集列组基本统计信息的 RUNSTATS 命令如下所示：

```
RUNSTATS ON TABLE DB2INST1.EMPLOYEE ON ALL COLUMNS AND COLUMNS((workdept, job));
```

这样，在查看 SQL 语句的访问计划，如下所示，估计的基数为 2.8，与实际行数非常接近。

```
      2.8
     TBSCAN
     (   2)
     7.65153
        1
        |
       42
 TABLE: DB2INST1
    EMPLOYEE
```

当多个列的数据具有相关性时，列组统计信息不仅能优化上述局部谓词的基数估计，也能对多个列连接谓词的基数进行更精确的估计。

注意：目前 DB2 还不支持为列组收集分布统计信息

列组统计信息在某些情况下能帮助优化器修正基数的估计。不过，DB2 还不支持为列组收集分布统计信息，因此，优化器在使用列组统计信息时假定数据是均匀分布的。

9.2.6 用优化概要定制访问计划

很多熟悉 Oracle 的读者都了解 Oracle 的 hint，它允许在 SQL 语句中插入相关的语法，从而定制 SQL 的访问计划。在 DB2 中，对应的是优化概要（Optimizer Guideline），也可以定制任何想要的访问计划。它提供了两种方式；一种是嵌入式，另一种是独立式。

1. 嵌入方式

将优化概要嵌入在 SQL 语句中，使用这种方式需要首先设置注册变量 DB2_OPTPROFILE=YES 并重启 DB2 服务器。如果需要在 SQL 调优的过程中对单条 SQL 语句尝试各种不同的访问计划，那么这种方式就很适合。

接下来用一个例子来说明。如下所示的 SQL 语句使用了嵌入式优化概要，它对优化器生成访问计划作出了这样的指示：

- 表的访问路径：访问 period 表使用索引 PERIOD_UIDX；访问 product 表使用表扫描；访问 sales 表使用索引扫描，由优化器根据成本模型为其选择合适的索引。
- 连接方式和顺序：period 表先与 product 做连接操作，连接方法为 NLJOIN；然后再与 sales 表连接，连接方法为 NLJOIN。

```
SELECT day,  prod_name,  sum(dollars)
FROM   product prd,  period per,  sales s
WHERE  prd.classkey = s.classkey and
       per.perkey = s.perkey and
       prd.prodkey = s.prodkey and
       prod_name like 'Darjeeling%' and
       day in ('SA', 'SU') and
       year = 2002
group by day, prod_name
```

```
/* <OPTGUIDELINES>
     <NLJOIN>
        <NLJOIN>
          <IXSCAN TABLE='per' INDEX='PERIOD_UIDX'/>
          <TBSCAN TABLE='prd'/>
       </NLJOIN>
       <IXSCAN TABLE='S'/>
     </NLJOIN>
</OPTGUIDELINES> */;
```

注意：优化概要中的表名

如果在 SQL 语句中对表名使用了别名，那么在优化概要中也必须使用别名，如上例中优化概要，用别名“S”表示 sales 表。

我们可以使用 db2exfmt 工具打印出这个 SQL 语句的访问计划，从下面可以看出：访问 period 表使用了索引扫描 PERIOD_UIDX，period 表先与 product 表做 NLJOIN，然后与 sales 再做 NLJOIN。优化概要为 sales 表指定了索引扫描但没有指定特定的索引，优化器根据成本模型最终选择了索引 ZZ_I1。

```
                              Rows
                             RETURN
                             (   1)
                              Cost
                               I/O
                               |
                             5.15254
                             GRPBY
                             (   2)
                             27910.8
                               151
                               |
                             5.15254
                             TBSCAN
                             (   3)
                             27910.7
                               151
                               |
                             5.15254
                             SORT
                             (   4)
                             27910.6
                               151
                               |
                             604.507
                             NLJOIN
                             (   5)
                             27882.2
                               151
                      /---------------+---------------\
                  9221.8                           0.0655519
                  NLJOIN                             FETCH
```

```
                       (   6)                              (  10)
                       14380.9                              10.4282
                         15                                1.06555
                 /---------+---------\                       /----+----\
           104.793                  88            0.0655519      69941
           FETCH                   TBSCAN           IXSCAN    TABLE: DB2ADMIN
           (   7)                  (   9)         (  11)           SALES
           136.917                 219.835          9.89349         Q3
              4                      11               1
        /---+----\                    |               |
      822         822               1298            69941
    IXSCAN   TABLE:DB2ADMIN   TABLE:DB2ADMIN   INDEX: DB2ADMIN
    (   8)        PERIOD          PRODUCT            ZZ_I1
    58.1848        Q4                Q5                Q3
       0
      |
      822
 INDEX: DB2ADMIN
   PERIOD_UIDX
      Q4
```

2. 独立方式

将优化概要嵌入到 SQL 语句中，在实际项目中并不是一个好的方式，因为数据是变化的，在某一时刻使用这个访问计划是最优的，在另一个时刻，却可能很差，这时再来修改应用中 SQL 语句的优化概要，很不方便。独立方式可以解决这个问题，它用独立的文件存储 SQL 语句和相应的优化概要，并注册到 DB2 中。这样，当需要改变访问计划时，不用修改 SQL 语句本身，只要修改独立文件的内容就行了。

不同于嵌入方式，使用独立方式时不需要设置注册变量 DB2_OPTPROFILE=YES，下面介绍使用独立方式的四个步骤。

（1）创建 OPT_PROFILE 表

OPT_PROFILE 表用于存储优化概要文件，可以调用 sysinstallobjects 存储过程自动创建或者使用 CREATE TABLE 语句手工创建。

如下所示，其中 schema 和 name 是组合主键，用来唯一确定某个优化概要。

```
call sysinstallobjects('opt_profiles', 'c', '', '');
```

或者

```
CREATE TABLE SYSTOOLS.OPT_PROFILE (
              SCHEMA  VARCHAR(128) NOT NULL,
              NAME    VARCHAR(128) NOT NULL,
              PROFILE BLOB (2M)    NOT NULL,
              PRIMARY KEY ( SCHEMA, NAME )
            );
```

（2）创建 OPT_PROFILE 文件

优化概要文件是一个标准的 XML 文件。如下所示，名为 opt_prof1.xml 的优化概要文件中，根节点为 OPTPROFILE，其下可以有多个 STMTPROFILE，每个 STMTPROFILE 指定一条 SQL

语句的优化概要。STMTPROFILE 主要由两部分组成，STMTKEY 包含 SQL 语句，而 OPTGUIDELINES 指定相应的优化概要，在 STMTKEY 中可以用属性 SCHEMA 指定 SQL 语句所引用表的默认模式，比如例子中指定默认模式为 DB2ADMIN。

```
<?xml version="1.0" encoding="UTF-8"?>
<OPTPROFILE>
  <STMTPROFILE ID="Q1">
    <STMTKEY SCHEMA="DB2ADMIN"><![CDATA[
    SELECT DAY,  prod_name, sum(dollars)
    FROM product prd, period per, sales s
    WHERE prd.classkey = s.classkey
      and per.perkey = s.perkey
      and prd.prodkey = s.prodkey
      and prod_name like 'Darjeeling%'
      and day in ('SA', 'SU')
      and year = 2002
    group by day, prod_name
    ]]></STMTKEY>
    <OPTGUIDELINES>
      <NLJOIN>
        <NLJOIN>
          <IXSCAN TABLE='per' INDEX='PERIOD_UIDX'/>
          <TBSCAN TABLE='prd'/>
        </NLJOIN>
        <IXSCAN TABLE='S'/>
      </NLJOIN>
    </OPTGUIDELINES>
  </STMTPROFILE>
</OPTPROFILE>
```

注意：优化概要文件的 SQL 语句

优化器使用 STMTKEY 的 SQL 语句匹配实际应用中的 SQL 语句，这种匹配能容忍大小写区别和空格多少，但其他方面必须一致。

（3）注册优化概要

注册优化概要，即将优化概要文件存储到第一步创建的 SYSTOOLS.OPT_PROFILE 表中。首先编辑如下内容，并保存到 opt_prof1.del 中。这里的内容与表 SYSTOOLS.OPT_PROFILE 结构一致，第一个值指定优化概要的模式 DB2ADMIN，第二个值指定优化概要的名字 OPT_PROF1，第三个值指定优化概要文件的名字，示例中为 opt_prof1.xml。

```
-- 文件 opt_prof1.del
"DB2ADMIN", "OPT_PROF1", "opt_prof1.xml"
```

然后用 import 命令将 opt_prof1.del 导入到表 SYSTOOLS.OPT_PROFILE 中，这样就完成了这个优化概要的注册，所使用的命令如下所示：

```
import from opt_prof1.del of del modified by lobsinfile
    insert into SYSTOOLS.OPT_PROFILE;
```

注册概要文件后，使用下面的 flush 命令刷新缓存中的优化概要。

```
flush optimization profile cache all;
```

（4）使用优化概要

最后，使用 set current optimization profile 命令设置当前要使用的优化概要。优化概要由 SYSTOOLS.OPT_PROFILE 中 schema 和 name 组合主键标识，示例中为 DB2ADMIN.OPT_PROF1，于是接下来的查询就能用到这个优化概要：

```
set current optimization profile = 'DB2ADMIN.OPT_PROF1';
set current schema db2admin;
set current explain mode explain;
SELECT day, prod_name, sum(dollars)
FROM product prd , period per, sales s
WHERE prd.classkey = s.classkey
  and per.perkey = s.perkey
  and prd.prodkey = s.prodkey
  and prod_name like 'Darjeeling%'
  and day in ('SA', 'SU')
  and year = 2002
group by day, prod_name;
set current explain mode no;
```

在 db2exfmt 打印的访问计划中，如下所示，其中有一段关于优化概要的信息，示例中这个 SQL 语句使用的优化概要为 DB2ADMIN.OPT_PROF1，而语句优化概要名字为 Q1，与 opt_prof1.xml 中 STMTPROFILE 的 ID 一致。

```
Profile Information:
--------------------
OPT_PROF: (Optimization Profile Name)
  DB2ADMIN.OPT_PROF1
STMTPROF: (Statement Profile Name)
  Q1

Original Statement:
------------------
SELECT ... ...
```

9.3 实战案例集锦

为了进一步帮助读者掌握 SQL 语句优化技巧并应用到实战中，本节将与读者分享我们在实际调优过程中分析问题和解决问题的过程和心得体会。每一个实战案例，都是由问题描述、诊断分析、解决方案共 3 部分内容组成的，建议读者阅读完问题描述和诊断分析后，先自己思考一下解决方案，随后再和书中提供的解决方案进行对比，这样学习起来印象更深刻，效果也更好一些。

9.3.1　案例 1：不可思议的物理读次数

在 SQL 语句优化过程中，要深入掌握物理设计、逻辑设计和数据库的配置参数等基本知识，因为很多 SQL 语句性能不佳是由于数据库设计或者配置不当导致的。

1. 问题描述

国内某银行客户报告了一个性能问题：他们的业务系统搭建在两套数据库上，一套是生产库，一套是复制库，两套数据库之间通过复制工具实现实时同步。但是，报表查询应用有关的 SQL 语句，在生产库上的响应速度很好，但在复制库上却很差。它们的具体性能参数如下所示，在复制库上运行的响应时间要比生产库多十几倍。

```
Server              Database              Execution time
-----------       ------------------     ----------------------------
Production          pmdb                 8 min 53 sec
Replication         pmrep               Around 123 min to 165 min
```

客户告诉我们，复制库与生产库的硬件配置完全一样，DB2 版本都是 V9.5，数据也完全一致，唯一区别在物理设计上：复制库使用的是自动存储表空间，生产库是非自动存储表空间。

2. 诊断分析

尽管客户声称唯一的区别为自动存储表空间，但还需要再次确认。让客户收集了 db2support 信息，发现这两个数据库环境有两点不同：

- 系统类型：操作系统均为 Linux，但是生产库为 390 平台，而复制库为 x86 平台。
- DB2 的版本：生产库为 V9.5.4，复制库为 V9.5.8。

下面是具体的诊断分析步骤。

（1）向客户提供一些建议

但都不奏效，例如收集列上的分布统计信息、为一些 SQL 语句新建组合索引、启用一些环境注册变量等。

（2）选择一条关键 SQL 语句作为切入点

由于报表查询应用涉及的 SQL 语句较多，为了加快问题的处理，从中选择一条关键 SQL 语句，请客户在复制库运行，在运行的过程中进行监控并收集快照信息。

（3）分析快照信息

在应用程序快照中，发现 CPU 的时间为 74.8 秒，只占整个运行时间很小一部分，这说明 CPU 不是问题。

但是，I/O 开销却很大，其中数据物理读次数为 103977，索引物理读次数高达 737988，均超过逻辑读次数的 10%。再结合数据库快照，发现索引物理读大部分是同步读，没有实现异步预取。所以，重点怀疑的就是物理读了。

应用程序快照和数据库快照信息如下所示：

```
            Application Snapshot

Application handle                          = 31776
Application status                          = UOW Executing
```

```
Buffer pool data logical reads             = 606274
Buffer pool data physical reads            = 103977

Buffer pool index logical reads            = 7672398
Buffer pool index physical reads           = 737988
Total buffer pool read time (milliseconds) = 5549225

Total Statement user CPU time              = 74.841954
Total Statement system CPU time            = 0.000000
Rows read                                = 1093028

          Database Snapshot

Buffer pool index logical reads            = 10070184
Buffer pool index physical reads           = 831266
Asynchronous pool index page reads          = 37734
```

（4）分析关键 SQL 语句的访问计划

访问计划中的相关片段如下所示，可以看到，这个访问计划用到了 5 个索引，并需要访问基本表 WR_PMT：

```
                                      0.0554048
                                        NLJOIN
                                        (  6)
                                        301863
                              /--------/        \-------\
                       72580.3                            7.63359e-07
                        FETCH                                FILTER
                        (  7)                                 (14)
                       270838                                21.302
    +---------+----------+-+---------+-----------+             |
  701259   1.31e+06    2.76596     1.4819     1.28818      1
    IOA    Table:       ISCAN       ISCAN      ISCAN      GRPBY
   (  8)   WR_PMT     (11)        (12)        (13)        (15)
  2899.33              21.304     21.3016     21.3013     21.3016
    |                      |          |          |           |
  701259             1.11487e+06  1.31e+06   1.11662e+06  1.18112
   SORT              Index:      Index:     Index:         ISCAN
   (  9)             IND_WP_06   IND_OR_04  PK_EM_01       (16)
  2899.33                                                  21.3014
    |                                                        |
  701259                                                  1.31e+06
   ISCAN                                                  Index:
   (10)                                                  IND_EIO_02
  2420.81
    |
1.11662e+06
Index:
IND_WP_05
```

而访问计划中这些表和索引的统计信息如下所示，表 WR_PMT 的总页数为 29597，比数据

物理读次数 103977 少得多，而 5 个索引（包括 IND_WP_05、IND_WP_06、IND_OR_04、PK_EM_01 和 IND_EIO_02）的页数总和都远远不及索引物理读的次数 737988。

```
Name  :        WR_PMT
Schema:         PM
              Number of Columns:                            58
              Number of Pages with Rows:                    29597
              Number of Pages:                              29597

IND_WP_05  Index Leaf Pages:                             993
IND_WP_06  Index Leaf Pages:                             546
IND_OR_04  Index Leaf Pages:                             8778

PK_EM_01   Index Leaf Pages:                             3281
IND_EIO_02 Index Leaf Pages:                            30591
```

（5）检查物理设计，包括表空间和缓冲池使用情况

顺藤摸瓜，继续查看相关表空间和缓冲池的定义，表 WR_PMT 的数据表空间和索引表空间分别为 TS_WP_03 和 IX_WP_03。

```
CREATE REGULAR TABLESPACE "TS_WP_03" IN DATABASE PARTITION GROUP IBMDEFAULTGROUP
       PAGESIZE 16384 MANAGED BY AUTOMATIC STORAGE
       AUTORESIZE YES
       BUFFERPOOL PM_BP08

CREATE REGULAR TABLESPACE "IX_WP_03" IN DATABASE PARTITION GROUP IBMDEFAULTGROUP
       PAGESIZE 16384 MANAGED BY AUTOMATIC STORAGE
       BUFFERPOOL PM_BP08

CREATE TABLE "PM    "."WR_PMT" ( ... )
             IN "TS_WP_03" INDEX IN "IX_WP_03" ;
```

这两个表空间均使用缓存池 PM_BP08，但它的尺寸在生产库和复制库上是不一样的，在生产库上有 64000 页。

```
--生产库 pmdb
CREATE BUFFERPOOL "PM_BP08"  SIZE 64000 PAGESIZE 16384;
```

但是，缓存池 PM_BP08 在复制库上的尺寸竟然只有 250 个页，这也太小了！

```
--复制库 pmrep
CREATE BUFFERPOOL "PM_BP08"  SIZE 250 PAGESIZE 16384;
```

3. 解决方案

解决方案水到渠成，在复制库上，调整 PM_BP08 缓冲池大小为 64000 页，这样就和生产库一样了，使用的语句如下所示：

```
ALTER BUFFERPOOL PM_BP08 IMMEDIATE SIZE 64000;
```

当然，如果 DBA 在缓冲池调优方面经验不足，建议启用 STMM 对其进行自动化调整。

心得分享： 对战争理论和实践缺乏基本了解，贸然上战场指挥战争，可能会导致全军覆没。同理，如果对调整缓冲池大小这样的基本功都没有掌握好，那么性能调优就无从谈起。读者读完这个案例后，应该反问自己：平时的功课是否做得足够好？

9.3.2 案例 2：理想的索引没用上

1. 问题描述

某著名投资银行报告了一个性能问题：根据查询条件创建了一个组合索引，但是从访问计划上看，这个索引没有用上，从而导致查询性能不高，竟然需要 6 分钟才能返回结果。

这条查询语句如下所示，查询条件使用到了字段 SYSTEM、TAG_DATE 和 TAG：

```
SELECT *
FROM AGDB.orders_history
WHERE system = 'jmb' AND tag_date = '08/06/2012' and tag = '927'
```

创建的组合索引如下所示，组合索引 ORD_HISTORY_IDXC 使用到了字段 SYSTEM、TAG_DATE、TAG 和 VERSION。这个组合索引完全匹配了查询中的 3 个谓词条件，而且这个索引是聚集索引（CLUSTER），应该是很理想的索引，但是该查询的访问计划没有用到这个索引，导致性能不高。

```
CREATE UNIQUE INDEX "AGDB    "."ORD_HISTORY_IDXC" ON "AGDB    "."ORDERS_HISTORY"
            ("SYSTEM" ASC,
             "TAG_DATE" ASC,
             "TAG" ASC,
             "VERSION" ASC)
            CLUSTER ALLOW REVERSE SCANS;
```

2. 诊断分析

下面是具体的诊断分析步骤。

（1）首先查看表 AGDB.ORDERS_HISTORY 和相关索引的定义：

```
CREATE TABLE "AGDB    "."ORDERS_HISTORY"  (
              "SYS_CATEGORY" SMALLINT NOT NULL ,
              "SYSTEM" VARCHAR(32) NOT NULL ,
              "TAG" VARCHAR(40) NOT NULL ,
              "TAG_DATE" DATE NOT NULL ,
              "VERSION" INTEGER NOT NULL ,
              "SIDE" SMALLINT ,
              "PROD_ID" VARCHAR(33) ,
              ... 省略 ...
              "CUSTODIAN_BANK" VARCHAR(256) ,
              "PEG_TYPE" SMALLINT )
             COMPRESS YES
             IN "AGG_D_TSP001" ;
CREATE INDEX "AGDB    "."ORD_HIS_INSERT_TIME" ON "AGDB    "."ORDERS_HISTORY"
            ("INSERT_TIME" ASC)
            ALLOW REVERSE SCANS;
```

```
    CREATE INDEX "AGDB   "."ORD_HIST_ID3" ON "AGDB   "."ORDERS_HISTORY"
                ("SYSTEM" ASC,
                 "TRANS_TIME" ASC)
                ALLOW REVERSE SCANS;
    CREATE INDEX "AGDB   "."ORD_HIST_ID4" ON "AGDB   "."ORDERS_HISTORY"
                ("TAG_DATE" ASC,
                 "PROD_ID" ASC)
                ALLOW REVERSE SCANS;
    CREATE UNIQUE INDEX "AGDB   "."ORD_HISTORY_IDXC" ON "AGDB   "."ORDERS_HISTORY"
                ("SYSTEM" ASC,
                 "TAG_DATE" ASC,
                 "TAG" ASC,
                 "VERSION" ASC)
                CLUSTER ALLOW REVERSE SCANS;
     ALTER TABLE "AGDB   "."ORDERS_HISTORY"
        ADD CONSTRAINT "ORDERS_H_PK1" PRIMARY KEY
                ("SYSTEM",
                 "TAG_DATE",
                 "TAG",
                 "VERSION");
```

（2）查看访问计划。

通过 db2exfmt 工具查看其访问计划：

```
    Original Statement:
------------------
SELECT *
FROM AGDB.orders_history
WHERE system = 'jmb' AND tag_date = '08/06/2012' and tag = '927'
    ... ...
    Access Plan:
-----------
   Total Cost:        22.7306
   Query Degree:      1

           Rows
          RETURN
          (   1)
           Cost
            I/O
             |
          7.9639
          FETCH
          (   2)
          22.7306
             3
        /---+----\
       8       9.81912e+08
     RIDSCN   TABLE: AGDB
     (   3)   ORDERS_HISTORY
     15.1481        Q1
       2
      |
```

```
      8
     SORT
    (   4)
    15.1478
      2
     |
      8
    IXSCAN
    (   5)
    15.1469
      2
     |
  9.81912e+08
INDEX: AGDB
 ORD_HIST_ID4
      Q1
```

从访问计划可以看到，DB2 优化器选择了索引 ORD_HIST_ID4(TAG_DATE，PROD_ID)用做索引扫描，但它只能用到 TAG_DATE 谓词，不能使用其他两个谓词 SYSTEM 和 TAG。这条查询竟然需要 6 分钟才返回结果！

从原理上看，组合索引 ORD_HISTORY_IDXC(SYSTEM,TAG_DATE,TAG,VERSION)更理想，完全匹配了查询中的 3 个谓词条件。

有趣的是，如果将 tag_date 上的值从“08/06/2012”改为“06/03/2012”，查询只需 0.2 秒就能返回结果。

```
SELECT *
FROM AGDB.orders_history
WHERE system = 'jmb' AND tag_date = '06/03/2012' and tag = '927'
```

（3）针对不同的 tag_date，对比它们的访问计划。

如果使用 TAG_DATE = '06/03/2012'这个谓词，优化器选择索引 ORD_HISTORY_IDXC（SYSTEM,TAG_DATE,TAG,VERSION）。从下面的访问计划可以看出，这个谓词过滤因子为 0.00689303，由于这个表是是 9 千多万行的大表，如果只用这个谓词过滤，得到的中间结果集大小估计为 9.81912e+08×0.00689303=6768348，依然很大，因此还需要结合其他两个谓词有效地过滤不需要的数据行。这种情况下，使用 ORD_HISTORY_IDXC 的性能更高。

```
3) Start Key Predicate
        Comparison Operator:          Equal (=)
        Subquery Input Required:       No
        Filter Factor:               0.00689303

        Predicate Text:
        --------------
        (Q1.TAG_DATE = '06/03/2012')

    3) Stop Key Predicate
        Comparison Operator:          Equal (=)
        Subquery Input Required:       No
        Filter Factor:               0.00689303
```

```
                Predicate Text:
                --------------
                (Q1.TAG_DATE = '06/03/2012')
```

如果使用 tag_date = '08/06/2012'这个谓词，优化器选择索引 ORD_HIST_ID4(TAG_DATE，PROD_ID）。从下面的输出可以看到这个谓词的过滤因子非常小，为 8.14737e-09，如果只用这个谓词过滤，得到的中间结果集大小估计为 9.81912e+08×8.14737e-09=8。这说明优化器认为单靠 tag_date 这个谓词就能有效地过滤掉不需要的行！从这点上看，采用 ORD_HIST_ID4 或 ORD_HISTORY_IDXC 并没有区别。

```
        3) Start Key Predicate
                Comparison Operator:           Equal (=)
                Subquery Input Required:        No
                Filter Factor:                 8.14737e-09

                Predicate Text:
                --------------
                (Q1.TAG_DATE = '08/06/2012')

         3) Stop Key Predicate
                Comparison Operator:           Equal (=)
                Subquery Input Required:        No
                Filter Factor:                 8.14737e-09

                Predicate Text:
                --------------
                (Q1.TAG_DATE = '08/06/2012')
```

再看两个索引的统计信息，发现 ORD_HIST_ID4 的层级为 3（B+树的层级），而 ORD_HISTORY_IDXC 是一个 4 级索引，这说明 ORD_HISTORY_IDXC 的索引树大很多。

如前面访问计划中的 I/O 成本所示，IXSCAN 的 I/O 成本为 2，FETCH 的 I/O 成为 3。由于 TAG_DATE = '08/06/2012' 具有非常低的过滤因子（8.14737e-09），优化器认为经过这个谓词过滤后只返回 8 行数据，已经足够将基本表的 I/O 降低到 1 个页面，使用索引 ORD_HIST_ID4 加上 2 个索引页 I/O，共 3 个 I/O。而使用索引 ORD_HISTORY_IDXC 可能需要 4 个 I/O 或者更多 I/O 成本，因此优化器认为采用 ORD_HIST_ID4 成本更低。

那么，为什么优化器的估计与实际性能不符呢？

进一步查看访问计划的对象详细信息，发现统计信息的更新日期是 06/31/2012，这在 08/06/2012 以前，也就是说 TAG_DATE = '08/06/2012'的数据并没有体现在统计信息中，于是优化器使用了不准确的统计信息计算出过滤因子为 8.14737e-09，从而导致优化器选择了错误的索引 ORD_HIST_ID4，合适的索引 ORD_HISTORY_IDXC 却没有用上。

```
    Objects Used in Access Plan:
    ---------------------------
    Schema:  AGDB
        Name:    ORDERS_HISTORY
    Type:    Table
            Time of creation:        2005-07-28-09.06.04.807513
```

```
        Last statistics update:   2012-06-31-04.10.23.193874
```

3. 解决方案

找到问题所在，解决方案其实很简单，使用下面的 RUNSTATS 命令更新这个表以及索引的统计信息即可，随后重新执行这个查询，响应时间降低到 0.2 秒：

```
runstats on table AGDB.orders_history with distribution and indexes all;
```

心得分享： 统计信息是优化器的“导航仪”，要想优化器正确工作，就需要及时更新统计信息。

9.3.3 案例 3：标记位（Flag）数据上的 runstats 时机

1. 问题描述

某客户在维护窗口运行批处理程序时，对数据库现有数据做了一些清理工作，然后执行 reorg 和 runstats 操作。全部执行完毕后，发现某条关键 SQL 语句的性能降低了：维护操作之前，该 SQL 语句的执行时间约为 12 秒，而维护操作之后，执行时间增加到 34 秒以上。

具体的性能数据由 db2batch 工具得出：

```
db2batch 测试的性能结果:运维操作之前
* Total Time:               11.953375 seconds
db2batch 测试的性能结果:运维操作之后
* Total Time:               34.430621 seconds
```

2. 诊断分析

下面是具体的诊断分析步骤。

（1）对比访问计划

维护操作前，这条 SQL 语句的访问计划如下所示：表 TRBK13 和表 TRFA01 之间使用 HSJOIN 做连接，HSJOIN 操作符的基数（返回行数）估计为 2383.39：

```
维护操作前的访问计划（性能好）:
                 ...
              2383.39
              ^HSJOIN
              (  18)
              197403
              110792
          /-----+------\
     332906         6083.32
     TBSCAN         TBSCAN
     (  19)         (  20)
     80566.9        116790
      45160          65632
       |              |
     665811         849703
 TABLE: DB2ADMIN  TABLE: DB2ADMIN
```

```
     TRBK13          TRFA01
       Q8              Q9
```

维护操作后，这条 SQL 语句的访问计划如下所示，表 TRBK13 和表 TRFA01 之间使用 NLJOIN 做连接，NLJOIN 操作符的基数（返回行数）估计为 0.007：

```
维护操作后的访问计划（性能差）:
              ...
           0.00715935
             ^NLJOIN
             (  18)
             80592.6
              45162
        /------+-------\
       1           0.00715935
    TBSCAN            FETCH
    (  19)            (  20)
    80566.9           38.5836
     45160               3
      |             /---+----\
    665810         1        849703
TABLE: DB2ADMIN  IXSCAN   TABLE: DB2ADMIN
   TRBK13      (  21)     TRFA01
      Q8        25.7288       Q9
                   2
                  |
                849703
            INDEX: SYSIBM
          SQL060502225749960
                 Q9
```

从访问计划中，可以看出运维前后，访问计划中基数差别巨大，这时有理由怀疑基数估计存在偏差，因此从这点入手分析问题。

（2）对比过滤因子

继续查看访问计划的详细信息输出，对比过滤因子，发现维护操作前后对“Q8.RONRI_SAKUZYO_FLG <> :?”这个谓词的过滤因子有着天壤之别，分别为 0.5 和 1.50592e-06，如下所示：

```
性能好的 HSJOIN:
             44) Sargable Predicate
                     Comparison Operator:           Not Equal (<>)
                     Subquery Input Required:        No
                     Filter Factor:                 0.5

                     Predicate Text:
                     --------------
                     (Q8.RONRI_SAKUZYO_FLG <> :?)
性能差的 NLJOIN:
            44) Sargable Predicate
                     Comparison Operator:           Not Equal (<>)
                     Subquery Input Required:        No
```

```
                    Filter Factor:              1.50592e-06

                    Predicate Text:
                    --------------
                    (Q8.RONRI_SAKUZYO_FLG <> :?)
```

（3）原来是标记位字段惹的祸

确认后，发现 Q8.RONRI_SAKUZYO_FLG 这个字段是个标记位，在维护操作前有 0 和 1 两个值，而维护操作后，根据业务逻辑的需要，所有的数据行全是 0。

实际上，当某个字段上所有数据都是同一个值时，DB2 估计等于（=）和不等于（<>）谓词的过滤因子是截然不同的：

- RONRI_SAKUZYO_FLG <> ?： DB2 会认为传入的参数值正好是列上的这个值，也就是没有行会满足条件，因此估计极低的过滤因子，从而估计很低的基数，接近于 0。在这个案例中，维护操作后，这个字段全部变成 0，优化器估计了极低的过滤因子 1.50592e-06，极低的基数 0.007。
- RONRI_SAKUZYO_FLG = ？：DB2 同样认为传入的参数值正好是列上的这个值，也就是所有行都满足条件，因此估计很高的过滤因子，接近于 1。

3. 解决方案

针对这个标记位的统计信息问题，有如下解决方案：

（1）使用 REOPT ALWAYS 选项重新绑定包含这条 SQL 语句的存储过程，命令如下所示（详情请参考 9.5.9 节）。这样，每次存储过程执行时，DB2 都会根据实际传入的值重新编译这条 SQL 语句，于是优化器就能生成正确的访问计划。

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'DB2ADMIN.TRBKPROC, 'REOPT ALWAYS')
```

当然，当这两个表的数据变化时，应该及时用 RUNSTATS 命令收集统计信息，使得优化器能作出正确的基数估计。

（2）使用如下优化概要定制访问计划，使用固定的 HSJOIN 连接这两个表。

```
/*<OPTGUIDELINES>
  <HSJOIN>
      <ACCESS TABID='Q8'/>
      <ACCESS TABID='Q9'/>
 </HSJOIN>
</OPTGUIDELINES>*/
```

（3）对这条 SQL 语句进行修改，使用 RONRI_SAKUZYO_FLG = ？代替原来的不等谓词 RONRI_SAKUZYO_FLG <> ？，并修改相关应用程序，使得调用该存储过程时传入相反的值。这样修改后，访问计划将会使用 HSJOIN，并且这个谓词的过滤因子将从 1.50592e-06 变为 1.0。

心得分享：标记位在数据建模中广泛应用，但标记位数据的统计信息收集需要格外留心，需要把握好时机，让标记位的统计信息正确反映实际的数据分布。

9.3.4　案例 4：NLJOIN vs HSJOIN 大比拼

1. 问题描述

某客户报告了一个 SQL 语句的性能问题，由于这条查询每天会执行几十万次，所以对整个系统的性能有较大影响。

先来看这条查询，它在 OA_MEMPRO 和 OA_NODE 这 2 个表上做连接，并不算复杂：

```
SELECT DISTINCT OA_MEMPRO.EXID, OA_MEMPRO.MEMBERID,
       OA_MEMPRO.MEMBERDISP, OA_MEMPRO.EMAIL,
       OA_MEMPRO.MEMBERTYPE, OA_MEMPRO.DYNAMIC,
       OA_MEMPRO.LOCALE, OA_MEMPRO.CREATED,
       OA_MEMPRO.LASTLOGIN,'joeOneIndex-16'
FROM ACTIVITIES.OA_MEMPRO OA_MEMPRO, ACTIVITIES.OA_NODE N1,
       ACTIVITIES.OA_NODE N2
WHERE N1.ACTIVITYUUID=N2.ACTIVITYUUID AND
       N2.CREATEDBY=OA_MEMPRO.MEMBERID AND
       OA_MEMPRO.MEMBERID!='003G091E0E4B47BEF6967B3131AD59007A53' AND
       N1.CREATEDBY='079G092D5B5E581369FD7FDA94EEFA00627C'
ORDER BY OA_MEMPRO.MEMBERDISP
```

在 V9.1 上，这条查询的访问计划采用 NLJOIN，并有很好的性能。但是升级到 V9.5 后，访问计划不再使用 NLJOIN，而是使用 HSJOIN，性能明显下降。

2. 诊断分析

下面是具体的诊断分析步骤。

（1）对比 V9.1 和 V9.5 下的访问计划

在 V9.1 上，使用 NLJOIN 的访问计划如下所示，NLJOIN 操作的成本为 3054.7。

```
                                Rows
                                RETURN
                                (   1)
                                 Cost
                                  I/O
                                  |
                                2892.01
                                TBSCAN
                                (   2)
                                3054.88
                                2052.7
                                  |
                                2892.01
                                SORT
                                (   3)
                                3054.87
                                2052.7
                                  |
                                2892.01
                                UNIQUE
```

```
                    (   4)
                    3054.71
                    2052.7
                      |
                    2892.01
                    ^NLJOIN
                    (   5)
                    3054.7
                    2052.7
               /--------+-------\
           1825.05            1.58462
           TBSCAN             FETCH
           (   6)             (  11)
           1033.71            15.1225
           136.702               2
             |              /----+---\
           1825.05      1.58462       83566
           SORT         IXSCAN   TABLE: ACTIVITIES
           (   7)       (  12)   OA_MEMPRO
           1033.7       7.56216        Q3
           136.702         1
             |             |
           1825.05       83566
           NLJOIN     INDEX: SYSIBM
           (   8)   SQL110316221740500
           1033.62         Q3
           136.702
        /------+------\
    67.3508          27.0976
    IXSCAN           IXSCAN
    (   9)           (  10)
    15.1245          15.1234
       2                2
      |                |
  3.55181e+06       3.55181e+06
INDEX: ACTIVITIES  INDEX: ACTIVITIES
  OA_IX_NODE_4      OA_IX_NODE_2
Q2                  Q1
```

在 V9.5 上，使用 HSJOIN 的访问计划如下所示，HSJOIN 操作的成本为 16179.9，大于在 V9.1 下的 NLJOIN 操作的成本：

```
                      Rows
                     RETURN
                     (   1)
                      Cost
                       I/O
                        |
                     1544.65
                     TBSCAN
                     (   2)
                     16181.6
                     2588.98
```

```
                             |
                           1544.65
                           SORT
                           (   3)
                           16181.4
                           2588.98
                             |
                           1544.65
                           ^HSJOIN
                           (   4)
                           16179.9
                           2588.98
                  /------------+-------------\
            365485                         1544.65
            FETCH                          NLJOIN
            (   5)                         (   7)
            14994.6                         1095.61
            2444.39                         144.583
       /----+----\                  /-------+-------\
      365486       365486       71.2915           21.6667
      IXSCAN   TABLE: ACTIVITIES  IXSCAN             IXSCAN
      (   6)   OA_MEMPRO   (   8)             (   9)
      973.779        Q3         15.2101           15.1752
      621.393                     2                 2
        |                         |                 |
      365486                  3.96802e+06       3.96802e+06
   INDEX: INSTLCA           INDEX: ACTIVITIES    INDEX: INSTLCA
 IDX104061828550000           OA_IX_NODE_4     IDX104061828510000
        Q3                        Q2                Q1
```

另外，在 V9.5 上，利用优化概要文件定制访问计划，并测试各种连接操作的性能如下，发现 NLJOIN 操作的性能要好于 HSJOIN。

```
NLJOIN - 0.39 seconds
HSJOIN - 1.73 seconds
MSJOIN - 1.66 seconds
```

（2）从索引入手分析

在 V9.5 上，为什么优化器没有选择性能更好的 NLJOIN 呢？这是问题的关键所在。再深入分析，发现很可能和 SQL110316221740500 这个索引有关，它是影响优化器选择 NLJOIN 和 HSJOIN 的关键因素。在 V9.1 的 NLJN 访问计划中，正是使用这个索引作为内表的索引扫描。

接下来，比较 SQL110316221740500 这个索引在 V9.1 和 V9.5 上的统计信息。

V9.1 上该索引的统计信息如下所示，索引 SQL110316221740500 具有极好的聚集度（clustering）和密度（density），均为 100。

```
    Schema: SYSIBM
    Name:     SQL110316221740500
    Type:     Index
           Time of creation:        2011-03-16-22.17.40.501173
           Last statistics update:
           Number of columns:        1
```

```
Number of rows:          83566
Width of rows:             -1
Number of buffer pool pages:     1741
Distinct row values:        Yes
Tablespace name:         OAREGTABSPACE
Tablespace overhead:        7.500000
Tablespace transfer rate:     0.060000
Source for statistics:        Single Node
Prefetch page count:        32
Container extent page count:     32
Index clustering statistic:     100.000000
Index leaf pages:          175
Index tree levels:          2
Index full key cardinality:     83566
Index first key cardinality:     83566
Index first 2 keys cardinality: -1
Index first 3 keys cardinality: -1
Index first 4 keys cardinality: -1
Index sequential pages:     175
Index page density:        100
Index avg sequential pages:     -1
Index avg gap between sequences:-1
Index avg random pages:     -1
Fetch avg sequential pages:     -1
Fetch avg gap between sequences:-1
Fetch avg random pages:     -1
Index RID count:         0
Index deleted RID count:     0
Index empty leaf pages:     0
Base Table Schema:        ACTIVITIES
Base Table Name:         OA_MEMPRO
Columns in index:
   MEMBERID
```

在 V9.5 上，使用 db2look 工具导出该索引的统计信息，如下所示，索引的聚集度和密度都很差，CLUSTERRATIO= 0.342662 和 DENSITY=42。聚集度和密度较低说明索引的页面在物理存储上比较分散，由于 NLJN 访问计划需要访问此索引，优化器认为读取该索引的页面需要更多的 I/O，因此成本较高。这是 V9.5 中优化器没有选择 NLJOIN 的原因。

```
UPDATE SYSSTAT.INDEXES
SET NLEAF=585,
   NLEVELS=2,
   FIRSTKEYCARD=365486,
   FIRST2KEYCARD=-1,
   FIRST3KEYCARD=-1,
   FIRST4KEYCARD=-1,
   FULLKEYCARD=365486,
   CLUSTERFACTOR=-1.000000,
   CLUSTERRATIO= 0.342662,
   SEQUENTIAL_PAGES=304,
   DENSITY=42,
   AVERAGE_SEQUENCE_GAP=0.000000,
```

```
    AVERAGE_SEQUENCE_FETCH_GAP=-1.000000,
    AVERAGE_SEQUENCE_PAGES=585.000000,
    AVERAGE_SEQUENCE_FETCH_PAGES=-1.000000,
    AVERAGE_RANDOM_PAGES=0.000000,
    AVERAGE_RANDOM_FETCH_PAGES=-1.000000,
    NUMRIDS=365486,
    NUMRIDS_DELETED=0,
    NUM_EMPTY_LEAFS=0,
    AVGPARTITION_CLUSTERRATIO=-1,
    DATAPARTITION_CLUSTERFACTOR=1.000000,
    INDCARD=365486
WHERE COLNAMES = '+MEMBERID'
      AND TABNAME = 'OA_MEMPRO' AND TABSCHEMA = 'ACTIVITIES';
```

3 解决方案

确定了根本原因后，解决这个问题就比较简单了，有如下解决方案：

（1）从访问计划的对比中可以看出，表 OA_MEMPRO 的基数在 V9.1 中显示为 83566，而在 V9.5 中则为 365486，发生了很大的变化。索引的统计信息也显示索引叶子页面数从 175 变为 585，但是索引的聚集度和密度变得很差。这说明 V9.1 上的统计信息已经过时，而升级到 V9.5，用户仅仅使用 RUNSTATS 更新了统计信息，但是没有做 REORG。

因此，升级到 V9.5 后，需要使用 REORG 命令重组这些表和表上的索引，并执行 RUNSTATS 重新收集统计信息，从而得到性能更好的 NLJOIN 访问计划。

（2）在 V9.5 上，可使用如下的优化概要文件，为该查询定制 NLJOIN 访问计划，从而获得所期望的性能。这样即使以后数据发生变化，该查询的访问计划也不会改变。

```
/*<OPTGUIDELINES>
<NLJOIN>
    <NLJOIN>
        <ACCESS TABID='Q2'/>
        <ACCESS TABID='Q1'/>
    </NLJOIN>
        <ACCESS TABID='Q3'/>
</NLJOIN>
</OPTGUIDELINES>*/
```

心得分享：哪怕一个不起眼的统计信息也会影响优化器所做的选择，从而导致大的性能差异。

9.3.5　案例 5：不是压缩惹的祸

1. 问题描述

客户报告了这样一个性能问题：在 DB2 V9.7 上执行一条查询语句，用到了两个表的连接操作，其中表 PS_PYMNT_VCHR_XREF 是大表，记录数为 6400000 行，另一个表 PS_PYCYCL_01_TMP6 是小表，记录数为 13678 行。如果对大表启用压缩，则这条语句执行起来需要 30 分钟；如果解除压缩，则该查询仅需 0.8 秒。

来看一下这个查询：

```
SELECT ... ...
FROM
  pqys.PS_PYMNT_VCHR_XREF A , pqys.PS_PYCYCL_01_TMP6 B
WHERE
    B.PAY_CYCLE = 'COMORA'
AND B.PAY_CYCLE_SEQ_NUM = 169
AND B.SOURCE_TXN = 'VCHR'
AND B.PYMNT_SELCT_STATUS NOT IN ('D','V')
AND A.BUSINESS_UNIT = B.BUSINESS_UNIT
AND A.VOUCHER_ID = B.PAY_DOC_ID
AND A.PYMNT_CNT = B.PAY_DOC_SEQ with CS;
```

使用 db2batch 测试这条查询在大表压缩和未压缩两种情况下的性能，如下所示：

```
对大表压缩：
* 5 row(s) fetched, 0 row(s) output.

* Prepare Time is:      0.023439 seconds
* Execute Time is:   1938.740764 seconds
* Fetch Time is:        0.000084 seconds
* Elapsed Time is:   1938.764287 seconds (complete)

对大表不压缩：
* 5 row(s) fetched, 0 row(s) output.

* Prepare Time is:      0.025302 seconds
* Execute Time is:      0.081455 seconds
* Fetch Time is:        0.000083 seconds
* Elapsed Time is:      0.106840 seconds (complete)
```

2. 诊断分析

下面是具体的诊断分析步骤。

（1）对比访问计划

对大表不压缩的访问计划如下所示，使用 NLJOIN 做连接操作：

```
Access Plan:
-----------
 Total Cost:   225837
 Query Degree:  1

                              Rows
                             RETURN
                              (   1)
                              Cost
                               I/O
                                |
                              13677
                             ^NLJOIN
                              (   2)
                             225837
```

```
                    139668
         /----------------+----------------\
       13677                                 1
      IXSCAN                               FETCH
      (   3)                               (   4)
      146.934                             22.7307
        83                                   3
        |                               /----+----\
       13677                          1          6.42044e+06
   INDEX: PQYS                      IXSCAN       TABLE: PQYS
  PSCPYCYCL_01_TMP6                 (   5)      PS_PYMNT_VCHR_XREF
        Q1                          15.163              Q2
                                      2
                                      |
                                  6.42044e+06
                                  INDEX: PQYS
                                PS_PYMNT_VCHR_XREF
                                      Q2
```

对大表压缩的访问计划如下所示，使用 HSJOIN 做连接操作：

```
Access Plan:
-----------
 Total Cost:   84942.9
 Query Degree:  1

                                Rows
                               RETURN
                               (   1)
                                Cost
                                 I/O
                                  |
                                13677
                               HSJOIN
                               (   2)
                               84942.9
                                55354
          /--------------------+-----------------\
      6.42044e+06                                  13677
         TBSCAN                                    IXSCAN
         (   3)                                    (   4)
         70278.6                                   146.934
          55271                                       83
           |                                          |
       6.42044e+06                                  13678
     TABLE: PQYS                                INDEX: PQYS
 PS_PYMNT_VCHR_XREF                           PSCPYCYCL_01_TMP6
          Q2                                         Q1
```

（2）对比语句快照

在大表未压缩的情况下，执行查询（NLJOIN）的语句快照如下所示：

```
Statement start timestamp                 = 10/04/2012 16:23:23.387481
Statement stop timestamp                  = 10/04/2012 16:23:23.704668
..
Total Statement user CPU time             = 0.042662
Total Statement system CPU time           = 0.000433
SQL compiler cost estimate in timerons     = 312278
SQL compiler cardinality estimate         = 13677
...
Rows read                               = 13684
...
Rows fetched                             = 5
Buffer pool data logical reads            = 1883
Buffer pool data physical reads           = 1
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads           = 41034
Buffer pool index physical reads          = 11
Buffer pool temporary index physical reads = 0
```

在大表压缩的情况下，执行查询（HSJOIN）的语句快照如下所示：

```
Statement start timestamp                 = 09/12/2012 13:43:39.839352
Statement stop timestamp                  = 09/12/2012 14:15:58.587003
...
Total Statement user CPU time             = 1231.349916
Total Statement system CPU time           = 0.007016
SQL compiler cost estimate in timerons     = 84943
SQL compiler cardinality estimate         = 13677
...
Rows read                               = 6420441
...
Rows fetched                             = 5
Buffer pool data logical reads            = 183439
Buffer pool data physical reads           = 0
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads           = 85
Buffer pool index physical reads          = 0
Buffer pool temporary index physical reads = 0
```

使用 HSJOIN 的访问计划用到的 CPU 时间为 1231 秒，超过 20 分钟。

（3）HSJOIN 连接字段类型不一致

于是进一步分析 HSJOIN 操作符的谓词，发现谓词 Q2.VOUCHER_ID = Q1.PAY_DOC_ID 竟然是 residual 谓词。在 4.5.2 节讲到，residual 谓词的处理性能是最差的，在使用 HSJOIN 连接后才需要计算这个谓词。

```
              2) Predicate used in Join
                    Comparison Operator:          Equal (=)
                    Subquery Input Required:       No
                    Filter Factor:               0.0078125

                    Predicate Text:
```

```
                --------------
                (Q2.PYMNT_CNT = Q1.PAY_DOC_SEQ)

        3) Residual Predicate
                Comparison Operator:           Equal (=)
                Subquery Input Required:        No
                Filter Factor:                 2.15222e-07

                Predicate Text:
                --------------
                (Q2.VOUCHER_ID = Q1.PAY_DOC_ID)

        4) Predicate used in Join
                Comparison Operator:           Equal (=)
                Subquery Input Required:        No
                Filter Factor:                 0.0769231

                Predicate Text:
                --------------
                (Q2.BUSINESS_UNIT = Q1.BUSINESS_UNIT)
```

最终，从表的定义中发现，字段 VOUCHER_ID 的类型为 CHAR(8)，而字段 PAY_DOC_ID 的类型为 CHAR(20)，尽管类型一致，但是长度不同。在 HSJOIN 中，要求连接谓词的两个字段类型（包括长度）完全一致，这便是问题的根源。

3. 解决方案

将字段 VOUCHER_ID 的类型定义从 CHAR(8)调整为 CHAR(20)，即和 PAY_DOC_ID 的类型以及长度完全一致，这样优化器就可以在访问计划中选择 HSJOIN 了。调整后，这个查询只需要 0.6 秒就能得到结果，比 NLJOIN 的 0.8 秒还要好！

> **心得分享**：连接谓词的类型匹配对 HSJOIN 非常重要。在一次国内 ERP 系统的 SQL 调优中，发现一个 VARCHAR(20)列与 CHAR(20)列做连接导致了严重的性能问题。将类型改成一致后，访问计划由 NLJOIN 改为了 HSJOIN，执行时间从 30 分钟以上减少到了 3 分钟以内。

9.3.6　案例 6：居高不下的 CPU 占用率

1. 问题描述

瑞士的一家银行客户对业务数据库进行了改造，添加了一些表，并导入了一些数据，重新执行过 reorg 和 runstats 后，按照计划上线后发现了一个性能问题：当系统忙时，db2sysc 进程的 CPU 占用率居高不下，响应速度也越来越慢。

2. 诊断分析

下面是具体的诊断分析步骤。

（1）首先对操作系统进行监控，获取性能数据。如下所示，CPU 占用率很高，而且主要

是用户 CPU（user cpu），占到 90%左右。

```
vmstats 命令的输出如下:
--
System configuration: lcpu=16 mem=32768MB ent=0.40

kthr    memory              page              faults              cpu
----- ----------- ------------------------ ------------ -----------------------
 r  b   avm    fre     re  pi  po  fr   sr  cy  in   sy  cs     us sy  id wa    pc    ec
78  0 6667113 743374   0   0   0   0    0   0  71  364428 178594 90  9   0   0  3.99 997.7
80  0 6667654 742827   0   0   0   0    0   0  80  443531 177407 87 13   0   0  3.99 997.2
80  0 6667150 743328   0   0   0   0    0   0  74  349676 161759 89 11   0   0  3.99 997.7
79  0 6667216 743258   0   0   0   0    0   0  66  382124 173518 90 10   0   0  3.99 997.7
80  0 6667368 743098   0   0   0   0    0   0  81  409346 140119 88 12   0   0  3.97 991.7
76  0 6667193 743270   0   0   0   0    0   0  51  371598 182898 89 11   0   0  3.99 996.9
78  0 6667149 743309   0   0   0   0    0   0  66  468944 197713 87 13   0   0  3.99 997.4
79  0 6667115 743338   0   0   0   0    0   0  95  405266 201333 87 13   0   0  3.98 994.7
80  0 6666900 743521   0   0   0   0    0   0 155  369984 179370 89 11   0   0  3.99 998.1
82  0 6667440 742976   0   0   0   0    0   0 138  376930 184825 88 12   0   0  3.96 990.3

nmon shows:
--
Time      PID       %CPU   %Usr %Sys Threads Size ResText ResData CharIO %RAM Paging Command
10:10:50 14286884 165.5   161.23 4.27 131    110180 140    110004 487194 0    112    db2sysc
10:11:50 14286884 185.83 180.32 5.51 131     110180 140    110004 368515 0    0      db2sysc
10:12:51 14286884 208.09 202.61 5.48 107    110180 140    110004 400880 0    0      db2sysc
10:13:51 14286884 162.5   158.15 4.35 107    110180 140    110004 359077 0    0      db2sysc
10:14:51 14286884 155.39 151.08 4.3   107    110180 140    110004 388350 0    0      db2sysc
10:15:51 14286884 119     115.33 3.67 128    110180 140    110004 413583 0    107    db2sysc
```

（2）分析数据库快照

如下所示，使用 CPU 最多的是一个名为 MDSECURE.GetAllBindsByUser 的存储过程调用，共发生 196 次调用，用户 CPU 时间占用 6847 秒，接近 2 个小时，平均每次调用耗时 71 秒。另外，由这个存储过程导致的索引读次数竟然达到了 4503695455 次！正是由于这么惊人的索引读次数导致了 CPU 使用率过高。

```
     Number of executions              = 196
     Number of compilations             = 1
     Worst preparation time (ms)         = 18
     Best preparation time (ms)          = 18
     Internal rows deleted              = 0
     Internal rows inserted             = 0
     Rows read                        = 1714
     Internal rows updated              = 0
     Rows written                     = 0
     Statement sorts                   = 132
     Statement sort overflows            = 0
     Total sort time                   = 0
     Buffer pool data logical reads      = 1716
     Buffer pool data physical reads     = 23
     Buffer pool temporary data logical reads   = 0
```

```
 Buffer pool temporary data physical reads  = 0
 Buffer pool index logical reads    = 4503695455
 Buffer pool index physical reads   = 11000
 Buffer pool temporary index logical reads  = 0
 Buffer pool temporary index physical reads = 0
 Buffer pool xda logical reads     = 0
 Buffer pool xda physical reads    = 0
 Buffer pool temporary xda logical reads   = 0
 Buffer pool temporary xda physical reads  = 0
 Total execution time (sec.microsec)= 5948.129121
 Total user cpu time (sec.microsec) = 6847.494026
 Total system cpu time (sec.microsec)= 32.953288
 Total statistic fabrication time (milliseconds) = 0
 Total synchronous runstats time  (milliseconds) = 0
 Statement text                  = call MDSECURE.GetAllBindsByUser(?,?)

... ...
Statement type                          = Static SQL Statement
Statement                              = Fetch
Section number                          = 1
Application creator                      = MDSECURE
Package name                           = P5595591
Consistency Token                        =
Package Version ID                       =
Cursor name                            = C1
Statement database partition number       = 0
Statement start timestamp                = 01/22/2013 10:12:32.362211
Statement stop timestamp                 =
Elapsed time of last completed stmt(sec.ms)= 0.000105
Total Statement user CPU time             = 145.796688
Total Statement system CPU time           = 10.135695
SQL compiler cost estimate in timerons     = 2
SQL compiler cardinality estimate         = 1


Buffer pool data logical reads            = 29
Buffer pool data physical reads           = 0
Buffer pool temporary data logical reads   = 0
Buffer pool temporary data physical reads  = 0
Buffer pool index logical reads           = 18631760
```

继续追踪 MDSECURE.GetAllBindsByUser 存储过程中定义游标 C1 的这条 SQL 语句，如下所示，这条 SQL 在表 desktops 和 bindings 之间做连接：

```
SELECT d_pmid, d_ip_address,  ... ,
      b_pmid, b_dt_type ...
FROM(
  SELECT d.pmid AS d_pmid, d.ip_address as d_ip_address, ... ,
        b.pmid AS b_pmid, b.dt_type AS b_dt_type, ...
  FROM MDSECURE.desktops d LEFT JOIN MDSECURE.bindings b
       ON (b.pmid = d.pmid)
  WHERE b.delete_flag IS NULL AND b.dt_type<>'T' AND b.userID = ?
  order by b.LASTMODIFIED desc) as cur
```

```
where 10 >
   (select count(*)
    from (
      SELECT d.pmid AS d_pmid, d.ip_address AS d_ip_address, ... ,
             b.pmid AS b_pmid, b.dt_type AS b_dt_type, ...
      FROM MDSECURE.desktops d LEFT JOIN MDSECURE.bindings b
           ON (b.pmid = d.pmid)
      WHERE b.delete_flag IS NULL AND b.dt_type<>'T' AND b.userID = ?
      order by b.LASTMODIFIED desc) as c
    where c.lastmodified > cur.lastmodified)
```

获取这条 SQL 语句的访问计划，如下所示，两个表上的扫描都用到了索引，总成本为366.358，这个值在合理范围内，访问计划也并没有什么不妥。

```
Access Plan:
-----------
Total Cost: 366.358
Query Degree: 1

                                    Rows
                                   RETURN
                                   (   1)
                                    Cost
                                     I/O
                                    |
                                   2.15702
                                   NLJOIN
                                   (   2)
                                   366.358
                                   61.2987
                        /------------+------------\
                    6.47106                      0.333333
                    ^NLJOIN                       FILTER
                    (   3)                       (  12)
                    252.578                       113.426
                    37.1859                       24.1128
               /------+-------\                      |
           6.47113         0.999989                  1
           TBSCAN           FETCH                  GRPBY
           (   4)           (  10)                 (  13)
           44.5358           32.16                 113.425
           17.6417          3.02021                24.1128
             |            /---+----\                 |
           6.47113    0.999989   7.34515e+06    3.23553
           SORT        IXSCAN   TABLE: MDSECURE  TBSCAN
           (   5)      (  11)     DESKTOPS       (  14)
           44.5349     21.3007       Q6          113.425
           17.6417        2                      24.1128
             |            |                        |
           6.47113    7.34515e+06                 3.23553
           FETCH     INDEX: MDSECURE               SORT
           (   6)     PK_DESKTOPS                 (  15)
           44.5323        Q6                      113.425
```

```
         17.6417                            24.1128
        /---+----\                            |
    6.48247    7.28217e+06                    3.23553
    RIDSCN   TABLE: MDSECURE                  ^NLJOIN
    (   7)     BINDINGS                      (  16)
    21.3052       Q7                        113.423
       2                                   24.1128
      |                             /---------+---------\
    6.48247                       3.23557             0.999989
    SORT                         FETCH                IXSCAN
    (   8)                       (  17)               (  21)
    21.3048                       44.5319              21.3007
       2                         17.6417                 2
      |                        /---+----\               |
    6.48247                 6.48247    7.28217e+06     7.34515e+06
    IXSCAN                  RIDSCN   TABLE: MDSECURE  INDEX: MDSECURE
    (   9)                  (  18)     BINDINGS       PK_DESKTOPS
    21.3039                 21.3052       Q2             Q1
       2                      2
      |                       |
   7.28217e+06               6.48247
 INDEX: MDSECURE              SORT
 BINDINGS_USERID              (  19)
      Q7                    21.3048
                              2
                             |
                           6.48247
                           IXSCAN
                           (  20)
                           21.3039
                              2
                             |
                          7.28217e+06
                        INDEX: MDSECURE
                        BINDINGS_USERID
                             Q2
```

（3）分析存储过程的访问计划

什么地方不对劲呢？怀着这种疑问，使用 db2expln 命令打印出了这个存储过程（包名为 P5595591）的访问计划，如下所示（关于打印方法，请参考 9.2.1 节）：

```
db2expln -d rsdb -c MDSECURE -p P5595591 -g -o sp.expln
```

于是，看到了最开始的那几行，寻思这 2012 年都快过去了，而这个存储过程的 bind 时钟还停留在 2010 年，如下所示。

```
******************** PACKAGE ***************************************

Package Name = "MDSECURE"."P5595591"  Version = ""

     Prep Date = 2010/02/27
     Prep Time = 17:54:46
```

```
    Bind Timestamp = 2010-02-27-17.54.46.372606
```

3. 解决方案

使用如下的 rebind 命令重新绑定这个存储过程，然后 CPU 使用率回到正常水平，一切归于平静！

```
rebind package MDSECURE.P5595591 resolve any;
```

或者

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MDSECURE.GetAllBindsByUser', 'ANY');
```

心得分享：数据库性能问题的外在表现与根源，有时似乎扯不上关系。可能是因为时间久了被忘了，也可能是用户不经意间执行了某个命令（例如，执行了 reorg 和 runstats，但是忘了执行 rebind），最终性能问题悄然而至。

9.4 高级 SQL 语句应用

作为数据库查询语言，SQL 语言的能力非常强大，除了提供基本的查询、增加、更新和删除功能语句外，还有一些高级功能。有时候需要多条基本 SQL 语句才能完成的功能，仅用一条高级 SQL 语句就非常方便地完成了，而且性能更高。本节将介绍一些这样的高级 SQL 语句，如果运用得当，必能在性能优化中发挥举足轻重的作用。

9.4.1 Merge 语句

在数据仓库应用中，很多开发人员在开发 ETL（Extract-Transform-Load 的缩写）脚本时，会遇到一种常见需求：将一个源表的数据合并到另一个目标表中。按照一般的思路，实现该需求至少需要 UPDATE 和 INSERT 两条 SQL 才能完成，而且性能也不怎么好。这个时候，就可以发挥 Merge 语句的威力了。

下面举一个数据仓库中的例子来说明，其中涉及的表有两个，分别是源表 purchase 和目标表 inventory，它们的 DDL 定义和初始化数据如下所示：

```
CREATE TABLE purchase(
    prod_id INTEGER NOT NULL PRIMARY KEY,
    prod_name VARCHAR(20) NOT NULL,
    price DECIMAL(10,2) NOT NULL,
    qty INTEGER NOT NULL);

CREATE TABLE inventory(
    prod_id INTEGER NOT NULL PRIMARY KEY,
    prod_name VARCHAR(20) NOT NULL,
    qty INTEGER NOT NULL);
```

```
INSERT INTO purchase (prod_id, prod_name, price, qty)
VALUES
   (101, 'IPad4',          1599.00,  2000),
   (103, 'IPhone5',        4299.00,  1800),
   (203, 'Galaxy Note II',  3699.00,  2500),
   (301, 'ThinkPad',       4688.00,  500);

INSERT INTO inventory (prod_id, prod_name, qty)
VALUES
   (101, 'IPad4',          800),
   (102, 'IPhone4S',       1800),
   (201, 'Galaxy S3',      2000),
   (301, 'ThinkPad',       300),
   (401, 'Kindle',         1400);
```

现在需要将 purchase 表的数据合并到 inventory 表，换句话说，如果 purchase 表的 prod_id 在 inventory 表中存在，则执行更新操作，即将 qty 值累加；否则，则向 inventory 表插入一条新纪录。

按照一般的思路，实现该需求至少需要 UPDATE 和 INSERT 两条 SQL 才能完成，示例如下，这两条 SQL 都包含子查询，看起来很别扭：

```
UPDATE inventory i SET qty = qty +
  (SELECT qty FROM purchase where prod_id  = i.prod_id)
WHERE i.prod_id IN (SELECT prod_id FROM purchase);

INSERT INTO inventory (prod_id, prod_name, qty)
  SELECT p.prod_id, p.prod_name, p.qty FROM purchase p
  WHERE prod_id NOT IN(SELECT prod_id FROM inventory);
```

Merge 语句的存在就是为了解决此类问题，实现“匹配则更新，不匹配则插入”的业务逻辑。如下所示，使用 Merge 语句，一条 SQL 就能解决问题，简洁高效：

```
MERGE INTO inventory i
     USING purchase p
ON (i.prod_id = p.prod_id)
WHEN MATCHED THEN
   UPDATE SET i.qty = i.qty + p.qty
WHEN NOT MATCHED THEN
   INSERT (prod_id, prod_name, qty)
   VALUES (p.prod_id, p.prod_name, p.qty);
```

上面的例子充分展示了 Merge 语句的能力。在 DB2 中，Merge 语句的完整语法如下所示，在匹配时（WHEN MATCHED）执行 UPDATE 或 DELETE，在不匹配时（WHEN NOT MATCHED）执行 INSERT：

```
MERGE INTO target_table
     USING source_table
ON search-condition
WHEN MATCHED [AND additional_condition] THEN
   UPDATE / DELETE
WHEN NOT MATCHED  [AND additional_condition] THEN
   INSERT;
```

注意：Merge 中源表和目标表

Merge 中源表和目标表的定义并不要求完全一致，上例中，purchase 和 inventory 表结构也不完全一致。另外，也不要求是基本表，可以是子查询或视图。

为了更深入理解 Merge 语句，下面再举几个例子。

（1）匹配时，更新目标表。

```
MERGE INTO inventory i
     USING purchase p
     ON (i.prod_id = p.prod_id)
WHEN MATCHED THEN
    UPDATE SET i.qty = i.qty + p.qty;
```

（2）匹配时，删除目标表中匹配的行。

```
MERGE INTO inventory i
     USING purchase p
     ON (i.prod_id = p.prod_id)
WHEN MATCHED THEN
    DELETE ;
```

注意：Merge 语句中的 DELETE

DELETE 语句只能删除目标表的记录，无法删除源表。在此例中，如果要删除 purchase 表的记录，可以将 purchase 作为 Merge 语句的目标表，或者直接用 DELETE 语句删除。

（3）不匹配时，向目标表插入新行。

```
MERGE INTO inventory i
     USING (SELECT prod_id, prod_name, qty FROM purchase) p
     ON (i.prod_id = p.prod_id)
WHEN NOT MATCHED THEN
    INSERT (prod_id, prod_name, qty)
    VALUES (p.prod_id, p.prod_name, p.qty);
```

（4）在匹配或不匹配子句中加入额外条件。

```
MERGE INTO inventory i
     USING purchase p
ON (i.prod_id = p.prod_id)
WHEN MATCHED AND i.qty <=500 THEN
    UPDATE SET i.qty = i.qty + p.qty
WHEN MATCHED AND (i.qty >500 or p.qty >=1000 ) THEN
    UPDATE SET i.qty = MIN(i.qty + p.qty, 2000)
WHEN NOT MATCHED THEN
    INSERT (prod_id, prod_name, qty)
    VALUES (p.prod_id, p.prod_name, p.qty)
ELSE IGNORE;
```

9.4.2　层次查询和 SQL 递归

很多用户在平时的工作中，经常遇到层次模型（hierarchical model，狭义上称为树型结构）

的数据，比如公司组织结构、产品分类等，那么怎么使用 SQL 语句对其进行查询呢？

答案是 SQL 递归，DB2 支持下面两种递归方式来实现层次查询：

- CONNECT BY：与 Oracle 兼容的递归语法，从 DB2 V9.7 开始支持。
- With 递归：用 With 子句实现递归，这是 SQL 标准的语法，Oracle 直到 11g R2 才开始支持 With 递归。

本节以某公司组织结构的层次数据为例，分别讲解上述两种递归方式的用法。如下所示，这个公司组织机构的数据存放在员工表 EMPLOYEE 中，下面是它的 DDL 定义，其中字段 MGRNO 用于构成层次模型，从下面的输出中可以看到最顶级的管理者为 Ginni。

```
CREATE TABLE HR.EMPLOYEE(
  EMPNO  INTEGER NOT NULL PRIMARY KEY,
  NAME   VARCHAR(10) NOT NULL,
  SALARY DECIMAL(9, 2) NOT NULL,
  MGRNO  INTEGER);

INSERT INTO HR.EMPLOYEE VALUES
  ( 1, 'Ginni',    90000,  NULL),
  ( 2, 'Tom',      60000,  1),
  ( 3, 'Kim',      60000,  1),
  ( 4, 'Johh',     50000,  1),
  ( 5, 'Mike',     42000,  2),
  ( 6, 'Smith',    40000,  2),
  ( 7, 'Kevin',    36000,  2),
  ( 8, 'Eileen',   34000,  5),
  ( 9, 'Joe',      33000,  3),
  (10, 'Henry',    33000,  3),
  (11, 'Steven',   32000,  9),
  (12, 'Jane',     30000,  9),
  (13, 'Hayes',    50000,  4),
  (14, 'Scott',    48000,  4),
  (15, 'Robert',   36000, 13),
  (16, 'Stone',    38000, 13);
```

1. 用 CONNECT BY 递归语句实现

从 DB2 V9.7 开始，可以使用 CONNECT BY 递归，但是需要设置注册变量 DB2_COMPATIBILITY_VECTOR=08，并重启数据库，如下所示。

```
db2set DB2_COMPATIBILITY_VECTOR=08
db2stop
db2start
```

注意：Oracle 兼容库和 Oracle 兼容语法

在 DB2 中，使用某些 Oracle 兼容语法并不需要重建数据库，只需设置 DB2_COMPATIBILITY_VECTOR 中相应的位（bit）并重启 db2 即可。但是如果要使用 DB2 支持的 Oracle 所有特性，特别是 Oracle 兼容的类型，需要设置 DB2_COMPATIBILITY_VECTOR=ORA 后重启 db2，然后新建的数据库才具有 Oracle 兼容特性。

先看一下 CONNECT BY 递归的语法格式，如下所示，其中 START WITH 用来指明递归的初始条件，CONNECT BY 子句表示递归关系，可以用 PIROR 关键字来指定上一层递归中的列，如 PRIOR EMPNO = MGRNO，即上一层递归中的 EMPNO 等于当前行的 MGRNO。

```
SELECT columns [, LEVEL, CONNECT_BY_ROOT, SYS_CONNECT_BY_PATH ]
FROM table_name
START WITH search-condition
CONNECT BY [ NOCYCLE ] PRIOR column = column
[ ORDER [ SIBLINGS ] BY ... ]
```

在 CONNECT BY 的层次查询中，还可使用如下伪列、函数和子句：

- LEVEL 伪列：节点在层次查询中的层级，根节点的 LEVEL 为 1。
- CONNECT_BY_ROOT 伪列： 层次查询中的根节点。
- SYS_CONNECT_BY_PATH 函数：根节点到当前节点的路径。
- ORDER SIBLINGS BY 子句：对同一父节点下的所有子节点按某字段排序。

下面举例说明。

（1）查询“Kim”经理下的所有员工，如下所示，使用 ORDER SIBLINGS BY 子句按照 EMPNO 字段排序：

```
 SELECT EMPNO, NAME, SALARY, LEVEL,
      CONNECT_BY_ROOT NAME AS ROOT,
      SUBSTR(SYS_CONNECT_BY_PATH(NAME, '/'), 1, 25) AS PATH
FROM HR.EMPLOYEE
START WITH NAME = 'Kim'
CONNECT BY PRIOR EMPNO = MGRNO
ORDER SIBLINGS BY EMPNO;
EMPNO      NAME      SALARY     LEVEL      ROOT      PATH
----------- ---------- ----------- ----------- ---------- -------------------------
         3 Kim          60000.00          1 Kim       /Kim
         9 Joe          33000.00          2 Kim       /Kim/Joe
        11 Steven        32000.00          3 Kim       /Kim/Joe/Steven
        12 Jane         30000.00          3 Kim       /Kim/Joe/Jane
        10 Henry        33000.00          2 Kim       /Kim/Henry

  5 record(s) selected.
```

（2）如果需要查询从“Ginni”经理开始，仅需 3 级组织结构并只返回 SALARY > 40000 的员工，所使用的语句如下所示，从输出可以看出 CONNECT BY 采用深度优先搜索：

```
SELECT EMPNO,  NAME, SALARY, LEVEL,
      CONNECT_BY_ROOT NAME AS ROOT,
      SUBSTR(SYS_CONNECT_BY_PATH(NAME, '/'), 1, 25) AS PATH
FROM HR.EMPLOYEE
WHERE SALARY > 40000 AND LEVEL < 4
START WITH NAME = 'Ginni'
CONNECT BY PRIOR EMPNO = MGRNO;

EMPNO      NAME      SALARY     LEVEL      ROOT      PATH
```

```
----------- ---------- ----------- ----------- ---------- -------------------------
         1 Ginni        90000.00           1 Ginni      /Ginni
         2 Tom          60000.00           2 Ginni      /Ginni/Tom
         5 Mike         42000.00           3 Ginni      /Ginni/Tom/Mike
         3 Kim          60000.00           2 Ginni      /Ginni/Kim
         4 Johh         50000.00           2 Ginni      /Ginni/Johh
        13 Hayes        50000.00           3 Ginni      /Ginni/Johh/Hayes
        14 Scott        48000.00           3 Ginni      /Ginni/Johh/Scott

  7 record(s) selected.
```

（3）查询员工“Stone”的经理，以及整个管理链，查询语句如下所示：

```
SELECT EMPNO, NAME, LEVEL,
      SUBSTR(SYS_CONNECT_BY_PATH(NAME, '/'), 1, 25) AS PATH
FROM HR.EMPLOYEE
START WITH NAME = 'Stone'
CONNECT BY EMPNO = PRIOR MGRNO
ORDER  BY LEVEL;

EMPNO      NAME      LEVEL      PATH
----------- ---------- ----------- -------------------------
        16 Stone              1 /Stone
        13 Hayes              2 /Stone/Hayes
         4 Johh               3 /Stone/Hayes/Johh
         1 Ginni              4 /Stone/Hayes/Johh/Ginni

  4 record(s) selected.
```

（4）如果需要查询“Kim”经理下面所有员工的薪金总和，语句如下所示：

```
SELECT SUM(SALARY) FROM
    (SELECT EMPNO,  NAME, SALARY
     FROM HR.EMPLOYEE
     START WITH NAME = 'Kim'
     CONNECT BY PRIOR EMPNO = MGRNO)
1
---------------------------------
                    188000.00

  1 record(s) selected.
```

2. With 递归

With 递归与 CONNECT BY 实现递归的基本原理是一样的，区别在于它使用 UNION ALL 来连接递归的初始条件和迭代步骤。

下面讲解如何使用 With 递归实现上面 CONNECT BY 语句同样的查询功能。

（1）查询经理“Kim”下的所有员工，语句如下所示：

```
WITH T(empno, name, salary, level, root, path) AS
 ( SELECT empno, name, salary, 1,
          name as root,
```

```
          varchar('/' || name, 25),
  FROM HR.EMPLOYEE
  WHERE NAME = 'Kim'   -- 递归初始条件，相当于 START WITH
 UNION ALL
  SELECT e.empno, e.name, e.salary,
         t.level + 1,                 -- 构建 LEVEL
         t.root,                      -- 构建 ROOT，相当于 CONNECT_BY_ROOT
         t.path || '/' || e.name      -- 构建 PATH，相当于 SYS_CONNECT_BY_PATH
  FROM T, HR.EMPLOYEE e
  WHERE t.empno = e.mgrno -- 递归步骤，相当于 CONNECT BY
 )
 SELECT * FROM T;

EMPNO       NAME       SALARY      LEVEL      ROOT       PATH
----------- ---------- ----------- ----------- ---------- -------------------------
SQL0347W  The recursive common table expression "DB2INST1.T" may contain an
infinite loop.  SQLSTATE=01605

         3 Kim          60000.00           1 Kim        /Kim
         9 Joe          33000.00           2 Kim        /Kim/Joe
        10 Henry        33000.00           2 Kim        /Kim/Henry
        11 Steven       32000.00           3 Kim        /Kim/Joe/Steven
        12 Jane         30000.00           3 Kim        /Kim/Joe/Jane

  5 record(s) selected with 1 warning messages printed.
```

注意：善意的无限循环警告

使用 With 递归时，DB2 可能会返回 SQL0347W 警告，这意味着递归公共表表达式可能包含无限循环。如果层次数据包含环（loop），那么递归中无限循环就有可能存在。因此，当收到 SQL0347W 这个善意警告时，需要检查一下是否会导致无限循环。

（2）如果需要查询从“Ginni”经理开始，只需 3 级组织结构并只返回 SALARY > 40000 的员工，所使用的语句如下所示，从输出可以看到 WITH 递归采用广度优先搜索：

```
WITH T(empno, name, salary, level, root, path) AS
 ( SELECT empno, name, salary, 1,
         name as root,
         varchar('/' || name, 25) as path
  FROM HR.EMPLOYEE
  WHERE NAME = 'Ginni'
 UNION ALL
  SELECT e.empno, e.name, e.salary,
         t.level + 1, t.root, t.path || '/' || e.name
  FROM T, HR.EMPLOYEE e
  WHERE t.empno = e.mgrno and e.SALARY > 40000 and level < 4
 )
 SELECT * FROM T;

EMPNO       NAME       SALARY      LEVEL      ROOT       PATH
----------- ---------- ----------- ----------- ---------- -------------------------
SQL0347W  The recursive common table expression "DB2INST1.T" may contain an
infinite loop.  SQLSTATE=01605
```

```
          1 Ginni         90000.00          1 Ginni      /Ginni
          2 Tom           60000.00          2 Ginni      /Ginni/Tom
          3 Kim           60000.00          2 Ginni      /Ginni/Kim
          4 Johh          50000.00          2 Ginni      /Ginni/Johh
          5 Mike          42000.00          3 Ginni      /Ginni/Tom/Mike
         13 Hayes         50000.00          3 Ginni      /Ginni/Johh/Hayes
         14 Scott         48000.00          3 Ginni      /Ginni/Johh/Scott

  7 record(s) selected with 1 warning messages printed.
```

比较：递归搜索的顺序

CONNECT BY 递归与 With 递归在搜索顺序的区别是：CONNECT BY 采用深度优先搜索，而 With 递归则是广度优先搜索。

（3）查询员工“Stone”的经理，以及整个管理链，查询语句如下所示：

```
WITH T(empno, name, mgrno, level, path) AS
 ( SELECT empno, name, mgrno,
         1, varchar('/' || name, 25) as path
   FROM HR.EMPLOYEE
   WHERE NAME = 'Stone'
  UNION ALL
   SELECT e.empno, e.name, e.mgrno,
         t.level + 1, t.path || '/' || e.name
   FROM T, HR.EMPLOYEE e
   WHERE e.empno = t.mgrno
 )
 SELECT empno, name, level, path FROM T;

EMPNO      NAME      LEVEL      PATH
----------- ---------- ----------- -------------------------
SQL0347W  The recursive common table expression "DB2INST1.T" may contain an
infinite loop.  SQLSTATE=01605

        16 Stone              1 /Stone
        13 Hayes              2 /Stone/Hayes
         4 Johh               3 /Stone/Hayes/Johh
         1 Ginni              4 /Stone/Hayes/Johh/Ginni

  4 record(s) selected with 1 warning messages printed.
```

（4）如果需要查询“Kim”经理下面所有员工的薪金总和，语句如下所示：

```
WITH T(empno, name, salary) AS
 ( SELECT empno, name, salary
   FROM HR.EMPLOYEE
   WHERE NAME = 'Kim'
  UNION ALL
   SELECT e.empno, e.name, e.salary
```

```
    FROM T, HR.EMPLOYEE e
    WHERE t.empno = e.mgrno
  )
  SELECT SUM(salary) FROM T
1
--------------------------------
                    188000.00

  1 record(s) selected.
```

9.4.3 报表开发中的 GROUP BY 扩展

GROUP BY 子句对指定字段进行分组，并对每一个分组计算聚集结果，例如 SUM、COUNT、AVG、MAX 等。然而，在 OLAP 应用中，经常需要对数据进行多维分析，不仅要计算基本分组的聚集结果，还需要计算不同维度的小计和总计。

针对上面的需求，用基本的 GROUP BY 和 UNION ALL 也可以实现，但是需要编写复杂的 SQL 语句，而且性能往往不佳。

幸运的是，DB2 提供了 GROUP BY 扩展，它很容易就能满足上面的需求。DB2 提供的 GROUP BY 扩展，包括 GROUP BY ROLLUP、GROUP BY CUBE、GROUP BY GROUPING SETS 以及 GROUPING 函数。

比较：GROUP BY 扩展

DB2 支持三种方式的 GROUP BY 扩展:

- GROUP BY ROLLUP：对基本分组计算聚集，然后依次计算小计，再计算总计。
- GROUP BY CUBE：对每个维度组合计算聚集，再计算总计。
- GROUP BY GROUPING SETS：对每个维度计算聚集结果。

在正式讲解 GROUP BY 扩展之前，需要使用示例数据，如下所示：

```
SELECT workdept, job, sex, sum(salary) sum_sal FROM employee
  GROUP BY workdept, job, sex
  ORDER BY workdept, job, sex;

WORKDEPT JOB      SEX SUM_SAL
-------- -------- --- ---------------------------------
A00      CLERK    F                        152750.00
A00      CLERK    M                         88500.00
A00      SALESREP F                         46500.00
A00      SALESREP M                         66500.00
D11      CLERK    F                         84630.00
D11      CLERK    M                        177880.00
D11      DESIGNER F                        233270.00
D11      DESIGNER M                        341100.00
D11      MANAGER  F                         96170.00
D11      MANAGER  M                         72250.00
E11      MANAGER  F                        398640.00
E11      MANAGER  M                        174425.00
```

```
E11      OPERATOR F                          154300.00
E11      OPERATOR M                           73090.00
E21      FIELDREP F                           35370.00
E21      FIELDREP M                          247150.00

  16 条记录已选择。
```

1. GROUP BY ROLLUP

GROUP BY ROLLUP（workdept，job）包含下面这些分组：

- （workdept，job）：按照这两个字段进行基本分组并计算聚集。
- （workdept）：在 workdept 字段上做小计，对应的数据行 job 字段为空。
- 计算总计，见最后一行。

可以看出，GROUP BY ROLLUP 先对基本分组进行聚集，然后依次小计，最后计算总计。因此，如果在 ROLLUP 中指定 n 列，根据这个分组过程，一共有 n+1 个分组，如下所示：

```
SELECT workdept, job, sum(salary) sum_sal FROM employee
   GROUP BY ROLLUP(workdept, job)
   ORDER BY workdept, job;

WORKDEPT JOB      SUM_SAL
-------- -------- ---------------------------------
A00      CLERK                        241250.00
A00      SALESREP                     113000.00
A00      -                            354250.00
D11      CLERK                        262510.00
D11      DESIGNER                     574370.00
D11      MANAGER                      168420.00
D11      -                           1005300.00
E11      MANAGER                      573065.00
E11      OPERATOR                     227390.00
E11      -                            800455.00
E21      FIELDREP                     282520.00
E21      -                            282520.00
-        -                           2442525.00

  13 条记录已选择。
```

2. GROUP BY CUBE

GROUP BY CUBE(workdept, job)包含下面这些分组：

- （workdept，job）：按照这两个字段进行基本分组并计算聚集。
- （workdept）：在 workdept 字段上做小计，对应的数据行 job 字段为空。
- （job）：在 job 字段上做小计，对应的数据行 workdept 字段为空。
- 计算总计：见最后一行。

可以看出，GROUP BY CUBE 对分组中的列进行组合，并计算所有组合分组的聚集结果。如果在 GROUP BY CUBE 中指定 n 列，根据这个分组过程，一共有 2 的 n 次方个分组，如下所示：

```
SELECT workdept, job, sum(salary) sum_sal FROM employee
   GROUP BY CUBE(workdept, job)
```

```
  ORDER BY workdept, job;

WORKDEPT JOB      SUM_SAL
-------- -------- --------------------------------
A00      CLERK                         241250.00
A00      SALESREP                      113000.00
A00      -                             354250.00
D11      CLERK                         262510.00
D11      DESIGNER                      574370.00
D11      MANAGER                       168420.00
D11      -                            1005300.00
E11      MANAGER                       573065.00
E11      OPERATOR                      227390.00
E11      -                             800455.00
E21      FIELDREP                      282520.00
E21      -                             282520.00
-        CLERK                         503760.00
-        DESIGNER                      574370.00
-        FIELDREP                      282520.00
-        MANAGER                       741485.00
-        OPERATOR                      227390.00
-        SALESREP                      113000.00
-        -                            2442525.00

 19 条记录已选择。
```

3. GROUP BY GROUPING SETS

GROUP BY GROUP SETS(workdept, job)包含下面这些分组：

- （workdept）：在 workdept 字段上做聚集，对应的数据行 job 字段为空。
- （job）：在 job 字段上做聚集，对应的数据行 workdept 字段为空。

可以看出，GROUPING SETS 只是对每个字段进行聚集计算。如果在 GROUPING SETS 中指定 n 列，根据这个分组过程，一共有 n 个分组方式，如下所示：

```
SELECT workdept, job, sum(salary) sum_sal FROM employee
  GROUP BY GROUPING SETS(workdept, job)
  ORDER BY workdept, job;

WORKDEPT JOB      SUM_SAL
-------- -------- --------------------------------
A00      -                             354250.00
D11      -                            1005300.00
E11      -                             800455.00
E21      -                             282520.00
-        CLERK                         503760.00
-        DESIGNER                      574370.00
-        FIELDREP                      282520.00
-        MANAGER                       741485.00
-        OPERATOR                      227390.00
-        SALESREP                      113000.00

 10 条记录已选择。
```

4. GROUPING 函数

从上面可以看到，GROUP BY 扩展生成的小计字段上的值为 NULL，这个 NULL 值是特殊的，但是原始数据本身也可能存在 NULL 值。如何区分它们呢？

这需要通过 GROUPING 函数来实现。GROUPING 以一个分组字段作为参数，如果输出是小计字段或者合计字段返回 1，否则返回 0。

如下所示，在 EMPLOYEE 表中，有一条记录的 workdept 值为 E21，job 值为 NULL，这样输出结果中倒数第 3 行（当前行输出）和倒数第 2 行（小计行输出）是一样的。有了 GROUPING 函数后，就可以根据 g_job 是否为 1 来判断，可以看出倒数第 2 行的 g_job 为 1，所以为小计行。

```
SELECT workdept, job, sum(salary) sum_sal,
       grouping(workdept) g_dept, grouping(job) g_job
   FROM employee
   GROUP BY ROLLUP(workdept, job)
   ORDER BY workdept, job;
WORKDEPT JOB      SUM_SAL                           G_DEPT     G_JOB
-------- -------- --------------------------------- ----------- -----------
A00      CLERK                        241250.00          0          0
A00      SALESREP                     113000.00          0          0
A00      -                           354250.00          0          1
D11      CLERK                        262510.00          0          0
D11      DESIGNER                     574370.00          0          0
D11      MANAGER                      168420.00          0          0
D11      -                          1005300.00          0          1
E11      MANAGER                      573065.00          0          0
E11      OPERATOR                     227390.00          0          0
E11      -                           800455.00          0          1
E21      FIELDREP                     282520.00          0          0
E21      -                            10000.00          0          0
E21      -                           292520.00          0          1
-        -                          2452525.00          1          1

  14 条记录已选择。
```

GROUP BY 扩展功能非常强大，理解它们的分组原理，掌握它们的功能，就能在实际报表开发工作中发挥重要作用。

9.5　编写高效 SQL 语句

SQL 语言非常灵活，可以使用多种写法实现同一功能，但是，这么多写法里面哪一种才是最高效的？或者换句话说，如何编写高效的 SQL 语句，从而充分利用 DB2 优化器的能力？本节将重点讲解这个问题。

9.5.1　不要检索多余的数据

在每条 SQL 语句的 SELECT 列表中只检索应用需要的列，把不必要的列放在 SELECT 列

表中将会消耗额外的内存和 CPU 资源。例如，很多开发人员喜欢使用“SELECT *”语句，它将检索所有列，并非好的 SQL 编程习惯。

对于数据行也是一样，应该尽量使用 WHERE 条件让 DB2 去过滤数据，而不是取出所有数据让应用程序来处理。同等情况下，从 DB2 传递到应用程序的数据行越少，SQL 效率就越高。开发人员容易犯的错误是：在应用程序中处理一些本来可以让 DB2 来过滤的数据。

9.5.2 避免在连接谓词中使用复杂表达式

连接谓词应该尽量简单，避免在连接谓词中使用表达式，最好是“字段名=字段名”，比如“T1.C1=T2.C2”，而且最好在设计数据模型时保持这两个字段的类型（包括类型长度或精度）完全一致，这样的连接谓词在执行时效率最高。

下面，用两个例子来说明。

（1）连接谓词使用表达式

例如：

```
WHERE SALES.PRICE * SALES.DISCOUNT = ORDERS.FINAL_PRICE
```

对于这条 SQL 语句，可以使用 4.2.1 节的技术在 SALES 表上加上冗余字段 FINAL_PRICE，并作为生成列由 PRICE×DISCOUNT 计算得到。

（2）连接谓词使用函数

例如：

```
WHERE UPPER(CUST.LASTNAME) = ORDERS.NAME
```

对于这条 SQL 语句，应该避免列 CUST.LASTNAME 和列 ORDERS.NAME 上的数据有大小写差别，这样就没有必要使用 UPPER 函数了。

注意：在 SQL 语句中使用复杂表达式

在 SQL 语句中使用含有复杂表达式的谓词有如下缺点：

- 影响优化器在生成访问计划时对索引和连接方法的选择。
- 影响优化器利用统计信息估计的精确度。
- 带有复杂表达式的谓词一般只能作为效率低下的 residual 谓词。
- 一般地，对这样的 SQL 应该做适当的改写。

9.5.3 将复杂表达式放在常量上

保持局部谓词足够简洁，这是一个重要原则。

如果确实需要使用函数或者表达式做转换，应该放在常量上，例如：“字段名=常数表达式”，这样便于匹配字段上的索引，也有利于优化器更准确地估算基数。

考虑下面的谓词，表达式作用在表的字段上：

```
EXPRESSION(Col) = 'constant'
INTEGER(TRANS_DATE)/100 = 200802
WHERE (CUST_ID * 100) + INT(CUST_CODE) = 123456
```

应该改写为如下形式，这样，如果字段 Col、TRANS_DATE 或者 CUST_ID 上有索引，优化器就能选择索引扫描：

```
Col = INVERS_EXPRESSION ('constant')
TRANS_DATE BETWEEN 20080201 AND 20080229
WHERE CUST_ID = 1234 AND CUST_CODE = '56'
```

9.5.4　使用 DB2 提供的日期类型

在数据库设计中，很多设计人员喜欢使用 INTEGER 或者 CHAR/VARCHAR 存储日期数据，这不是好的设计，实际上，直接使用 DB2 的日期类型是更好的选择。这样一方面可以利用 DB2 提供的大量日期函数；另一方面 DB2 优化器也可以将这些日期函数自动转换成简单谓词。

例如，在表 tpcds.date_dim 的 d_date 字段（类型为 DATE）上建立索引，这样对 WHERE 条件中使用日期函数 YEAR 和 MONTH 的查询，DB2 优化器能将这两个日期函数转换成简单谓词，而不是效率低下的 residual 谓词，从而提高查询性能，如下所示：

```
Original Statement:
------------------
select d_date_sk
from tpcds.date_dim
where year(d_date)= 2012 and month(d_date) =10

Optimized Statement:
-------------------
SELECT Q1.D_DATE_SK AS "D_DATE_SK"
FROM TPCDS.DATE_DIM AS Q1
WHERE ('2012-10-01' <= Q1.D_DATE) AND (Q1.D_DATE < '2012-11-01')

Access Plan:
-----------
     Total Cost:       15.1672
     Query Degree:         1

              Rows
             RETURN
             (   1)
              Cost
               I/O
               |
             29.9908
             FETCH
             (   2)
             15.1672
               2
          /----+-----\
       29.9908        73049
       IXSCAN   DP-TABLE: TPCDS
       (   3)       DATE_DIM
       7.57442         Q1
          1
         |
```

```
        73049
 DP-INDEX: TPCDS
      D_DATE
        Q1
```

9.5.5 谨慎对待隐式类型转换

DB2 V9.7 之前，DB2 一直遵循 SQL 标准的数据类型系统，遵守强类型运算规则，错误的数据类型运算将导致编译错误，比如字符串和数字不能比较，除非经过显式的类型转换。

DB2 V9.7 开始支持隐式数据类型转换（也叫自动类型转换）。如下所示，DB2 将字符类型自动转换成数值类型，也可以将数值类型转换成字符类型，还可以完成 DATE 类型与字符类型的相互转换。

```
-- C1 为数值类型，DB2 自动将'12'这个字符类型转换成数值类型，然后进行比较
SELECT sum(qty) FROM sales WHERE c1 >= '12';
```

隐式类型转换给开发人员一些便利，让他们不需花费精力去关注这些细枝末节，而由 DB2 完成数据类型自动转换。

隐式数据类型转换虽然很方便，却有诸多缺点：

- 对于错误的隐式类型转换，在编译时不会发现问题，而是到了运行时才报错，这会给查找和解决问题带来很大的困难。
- 从性能方面考虑，隐式数据类型转换虽然方便，但也是要消耗时间的，如果在循环中频繁地进行这种隐式类型转换，会消耗一定的 CPU 资源。
- 在查询中使用隐式类型转换可能导致索引抑制，从而带来性能上的损害。在 9.3.5 节的案例中，含有隐式转换的连接谓词 Q2.VOUCHER_ID = Q1.PAY_DOC_ID，即将 CHAR(8)的字段与 CHAR(20)字段进行比较，导致了性能问题。

因此，开发人员在使用隐式类型转换时应保持谨慎态度，只有真正理解其对 SQL 语句的性能影响后，才可以考虑使用。

9.5.6 外连接的顺序

对于内连接（INNER JOIN）而言，DB2 能自动选择和调整表与表连接的顺序，但是外连接会限制连接的顺序，从而导致一些好的访问计划无法生成。这里讲到的外连接包括 3 种：左外连接（LEFT OUTER JOIN）、右外连接（RIGHT OUTER JOIN）和全连接（FULL OUTER JOIN）。

下面来看某银行的一个实际案例，如下所示，4 个表之间的连接操作，T1 和 T2 先做 LEFT JOIN，再和 T3 做内连接，最后和 T4 做内连接。根据实际情况，T1 和 T3 之间的内连接谓词具有最好的过滤因子，但是由于 LEFT JOIN 存在，T1 不能与 T3 先做连接，而是必须先与 T2 做连接，才能与 T3 做内连接，由于 T1 是几亿行的大表，这样连接的效率非常低。

```
SELECT  ... ...
FROM ECRM.PARS_REC_APPLY_INFO T1
      LEFT JOIN ECRM.PARS_REC_APPROVE T2
          ON T1.NUM = T2.NUM
      INNER JOIN ECRM.ECR_CUST_MANAGER T3
          ON T1.PAYER_CUST_MANAGER_ID = T3.CUST_MANAGER_ID
```

```
        INNER JOIN AP_USER T4
            ON T1.ELEC_MAIL = T4.LOGIN_NAME||'@bank.com'
WHERE T4.USER_CODE = '9666196'
```

这时，必须手工调整表连接的顺序，让 T1 先和 T3 做内连接，再和 T4 做内连接，最后再与 T2 做左外连接，修改后的语句如下所示，根据测试结果，其性能提升了 2000 倍左右。

```
SELECT  ... ...
FROM ECRM.PARS_REC_APPLY_INFO T1
        INNER JOIN ECRM.ECR_CUST_MANAGER T3
            ON T1.PAYER_CUST_MANAGER_ID = T3.CUST_MANAGER_ID
        INNER JOIN AP_USER T4
            ON T1.ELEC_MAIL = T4.LOGIN_NAME||'@bank.com'
        LEFT JOIN ECRM.PARS_REC_APPROVE T2
           ON T1.NUM = T2.NUM
WHERE T4.USER_CODE = '9666196'
```

注意：外连接中表顺序

关于多表之间内连接的顺序，优化器会合理选择和调整内连接的顺序以获得最好性能。 但是，外连接是特殊的，它限制了连接的顺序，在编写多表连接的 SQL 语句时，应特别注意外连接时表的顺序。

9.5.7　IN 和 EXISTS 子查询

在 SQL 语言中，当一个查询语句嵌套在另一个查询中时，称为子查询。DB2 中的子查询有多种形式，这里主要讨论如下几种：

- IN 和 EXISTS；
- NOT IN 和 NOT EXISTS；
- 使用反连接（ANTI-JOIN）优化子查询。

1. IN 和 EXISTS

DB2 优化器在处理 IN 和 EXISTS 子查询时，会将其转换为连接，从而生成更优的访问计划。例如，下面的 IN 和 EXISTS 子查询是等价的：

```
select t1.c1, t1.c2, t1.c3
from T1
where exists(select 1 from T2 where T1.c1 = T2.c1)
或者
select t1.c1, t1.c2, t1.c3
from T1
where c1 in (select t2.c1 from T2)
```

DB2 优化器会将上面两条 SQL 语句转换为如下的连接方式，这样优化器在生成访问计划时就可以为下面的连接考虑多种连接方法，特别在这两个表还与其他表做连接时，这种转换将显得更有效。

```
select distinct t2.c1, t1.c1, t1.c2, t1.c3
from T1, T2
```

```
where T1.c1 = T2.c1
```

注意其中的 distinct 关键字，这是保持语义等价所必需的。显式的 DISTINCT 需要排序来实现，是一种成本较高的操作。只有当 T2.C1 字段具有唯一性（例如是主键或者字段上创建有唯一索引）时，优化器才不会在转换后的 SQL 语句中加 DISTINCT。因此，出于性能方面的考虑，应避免使用子查询，特别是嵌套层次较多的子查询。

注意：子查询（IN 和 EXISTS）与连接

在编写 SQL 语句，应尽量使用连接，其次才考虑子查询。

2. NOT IN 和 NOT EXISTS

一般情况下，NOT EXISTS 比 NOT IN 的性能要好，但是这并不绝对。

来看一个例子，下面的 SQL 语句使用 NOT IN 子查询列出所有还没有下过订单的客户。从访问计划图中可以看出，NOT IN 的访问计划执行顺序为：先扫描内表 STORE_SALES，然后再扫描外表 CUSTOMER，总成本为 7.42152e+06。

```
select * from tpcds.customer
where C_CUSTOMER_SK NOT IN
   (select SS_CUSTOMER_SK from tpcds.store_sales)

Access Plan:
-----------
       Total Cost:            7.42152e+06
       Query Degree:           1

                   Rows
                  RETURN
                  (   1)
                   Cost
                    I/O
                    |
                  250000
                  FETCH
                  (   2)
                7.42152e+06
                  8338.35
         +------------++------------+
       500000       413696      500000
       IXSCAN       TBSCAN   TABLE: TPCDS
       (   3)       (   4)     CUSTOMER
       1087.56      37920.1        Q3
       533.125      4929.22
         |            |
       500000       413696
   INDEX: SYSIBM     SORT
 SQL120728090612660  (   5)
         Q3         37861.1
                    4929.22
                      |
```

```
                    1e+07
                   IXSCAN
                   (   6)
                   25462.9
                   4929.22
                     |
                    1e+07
              DP-INDEX: TPCDS
                   SS_CUST
                     Q1
```

用 NOT EXISTS 改写上面的查询，相应的 SQL 语句和访问计划如下所示。访问计划的总成本为 37094.4，比 NOT IN 访问计划的成本要小得多。从访问计划图中可以看出，NOT EXISTS 的访问计划执行顺序为：先扫描外表 CUSTOMER，对于外表的每一行，再扫描内表 STORE_SALES，看是否满足 NOT EXISTS 谓词。

```
select * from tpcds.customer
where NOT EXISTS
   (select 1 from tpcds.store_sales C_CUSTOMER_SK=SS_CUSTOMER_SK);
Access Plan:
-----------
        Total Cost:            37094.4
        Query Degree:           1

               Rows
              RETURN
              (   1)
               Cost
                I/O
                |
               86304
             x^NLJOIN
             (   2)
              37094.4
               6344
        /------+-------\
     500000              20
     TBSCAN            IXSCAN
     (   3)            (   4)
     3690.2            60.9982
      2880                8
       |                 |
     500000             1e+07
 TABLE: TPCDS     DP-INDEX: TPCDS
    CUSTOMER           SS_CUST
       Q2                Q1
```

从上面的访问计划中，可以发现 DB2 处理这两种子查询机制上的差别：

- NOT IN 是自内向外的操作，即先得到子查询的结果，再执行外层的查询。
- NOT EXISTS 恰好相反，是自外向内的操作，先扫描外表，根据外表的每一行去扫描子查询中的内表。

在这个例子中，子查询的内表 STORE_SALES 是 1 千万行的大表，得到的 NOT IN 列表比较大，为 413696 个值（去重以后），在这种情况下，NOT IN 的性能要比 NOT EXISTS 的性能差很多。

一般地，如果子查询的结果集比较小，例如内表上有过滤因子很小的局部谓词时，可以选择使用 NOT IN 子查询，如下所示：

```
SELECT T1.* FROM T1 WHERE T1.C1 NOT IN
                 (SELECT 1 FROM T2 WHERE T2.C2 = 5);
```

相反的，如果外层查询的结果集比较小，例如外表上有过滤因子很小的局部谓词时，可以选择使用 NOT EXISTS 子查询，如下所示：

```
SELECT T1.* FROM T1 WHERE T1.C2 = 6 WHERE NOT EXISTS
                 (SELECT 1 FROM T2 WHERE T2.C1=T1.C1);
```

注意：NOT IN 和 NOT EXISTS 并不完全等价

NOT IN 和 NOT EXISTS 的语义并不完全等价，当外表的连接列可以为空值时，其查询结果可能不同，如下例所示:

```
with t1(c1,c2) as (values (1,1),(null,1)),
     t2(c1,c2) as (values (2,1))
select * from t1 where c1 not in (select c1 from t2)

C1          C2
----------- -----------
          1           1

  1 record(s) selected.

with t1(c1,c2) as (values (1,1),(null,1)),
     t2(c1,c2) as (values (2,1))
select * from t1 where not exists (select 1 from t2 where t1.c1= t2.c1)

C1          C2
----------- -----------
          1           1
          -           1

  2 record(s) selected.
```

3. 反连接

读者在阅读上面 NOT EXISTS 子查询的访问计划时，会发现连接操作 x^NLJOIN 与普通 NLJOIN 操作符不一样，有一个特殊的前缀。进一步查看这个操作符的详细信息，会发现 OUTERJN 这个参数被设置为 LEFT(ANTI)，如下所示：

```
    2) NLJOIN: (Nested Loop Join)
           Cumulative Total Cost:         37094.4
           Cumulative CPU Cost:           1.41028e+10
           Cumulative I/O Cost:           6344
```

```
                Cumulative Re-Total Cost:       7716.39
                Cumulative Re-CPU Cost:        1.35198e+10
                Cumulative Re-I/O Cost:        0
                Cumulative First Row Cost:      15.2109
                Estimated Bufferpool Buffers:   6344

                Arguments:
                ---------
                EARLYOUT: (Early Out flag)
                       LEFT
                FETCHMAX: (Override for FETCH MAXPAGES)
                       IGNORE
                ISCANMAX: (Override for ISCAN MAXPAGES)
                       IGNORE
                OUTERJN : (Outer Join type)
                       LEFT (ANTI)

                Predicates:
                ----------
                2) Predicate used in Join,
                       Comparison Operator:          Equal (=)
                       Subquery Input Required:       No
                       Filter Factor:                2e-06

                       Predicate Text:
                       --------------
                       (Q2.C_CUSTOMER_SK = Q1.SS_CUSTOMER_SK)
```

那么，什么是反连接？顾名思义，反连接是连接的反向操作，它的结果集是左外连接的结果集减去内连接的结果集。

```
T1 ANTIJOIN T2 = (T1 LEFT OUTER JOIN T2) - (T1 INNER JOIN T2)
```

在访问计划中，ANTIJOIN 是普通连接操作的特殊形式（有一个“x^”前缀），其实际操作符可以为 NLJOIN、MSJOIN 或 HSJOIN，只是将 OUTERJN 这个参数设置为 ANTI 表示该连接为反连接。

反连接（ANTI-JOIN）是 DB2 优化 NOT EXISTS 和 NOT IN 子查询的一种比较高效的方法，DB2 优化器会尝试将 NOT EXISTS 和 NOT IN 转换成反连接。

如果要对某个查询启用反连接，可以使用如下的优化概要来指导优化器做这种转换：

```
--将 NOT IN 转换成反连接的优化概要
<OPTGUIDELINES><NOTIN2AJ OPTION='ENABLE'/></OPTGUIDELINES>

--将 NOT EXISTS 转换成反连接的优化概要
<OPTGUIDELINES><NOTEX2AJ OPTION='ENABLE'/></OPTGUIDELINES>
```

如果要在整个实例范围内设置反连接，可以使用注册变量 DB2_ANTIJOIN，它的值可以为 YES、EXTEND 和 NO：

- 当 DB2_ANTIJOIN=YES：DB2 优化器将尝试将 NOT EXISTS 子查询转换为高效的反连接，单分区数据库上，默认为 YES。

- 当 DB2_ANTIJOIN=EXTEND：DB2 优化器将尝试将 NOT EXISTS 和 NOT IN 子查询都转换为反连接。
- 当 DB2_ANTIJOIN=NO：DB2 优化器将限制某些将 NOT EXISTS 子查询转换为反连接的机会。

然而，将 NOT EXISTS 和 NOT IN 转换成反连接并不能 100%保证能提高性能（尽管大多数情况能提高性能），因此应该做好全面的性能测试和验证。

9.5.8 OFNR 和 FFNR 子句

OFNR（OPTIMIZE FOR N ROWS）子句告诉优化器，返回整个结果集，但希望以最快的速度返回前 N 行。这时，DB2 会选择返回前 N 行结果最快的访问计划，但并不一定是获取整个结果集最佳的访问计划。

FFNR（FETCH FIRST N ROWS ONLY）子句告诉优化器，不用返回整个结果集，只返回前 N 行就可以了，这时 DB2 会选择获得整个结果集最佳的访问计划，但只返回前 N 行给应用。

SQL 开发人员应根据需要灵活选择 OFNR 或者 FFNR。

9.5.9 动态 SQL vs 静态 SQL

在 DB2 中，存在两种类型的 SQL 语句：动态 SQL 和静态 SQL。动态 SQL 的执行计划需要每次执行时编译；静态 SQL 语句的访问计划在绑定（BIND）时就已经确定，执行时只需要把访问计划调出来直接执行即可。

DB2 有一个动态语句高速缓存，它将最近执行的动态 SQL 语句和相应的访问计划缓存起来，如果下次遇到到同样的 SQL，直接从缓存中读取相应的访问计划，从而避免重复编译。

（1）对 OLTP 应用，尽可能减少编译时间。

在 OLTP 应用中，一条 SQL 的执行时间一般很短，利用动态语句高速缓存节省的编译时间和 CPU 开销相对来说是可观的，能带来性能的显著提升。

DB2 采用 HASH 算法判断一个动态 SQL 语句的执行计划是否在包缓存中。该算法根据 SQL 语句的文本计算 HASH 值，即使两个语句只有一个字母大小写的不同，也会造成 HASH 值不同，从而认为是两个不同的 SQL 语句。

例如，下面这两个语句会被看作是不同的动态 SQL：

```
SELECT EMP_ID, EMP_NAME, AGE FROM MANAGER_INFO WHERE EMP_ID = 918233
SELECT EMP_ID, EMP_NAME, AGE FROM MANAGER_INFO WHERE EMP_ID = 920122
```

可以考虑在应用程序程序中改写上述语句，将常量改写成参数标记，并在执行时将参数绑定为实际的值。这样，DB2 接收到的 SQL 语句是一样的，能够命中动态语句高速缓存，从而减少编译开销。

```
SELECT EMP_ID, EMP_NAME, AGE FROM MANAGER_INFO WHERE EMP_ID = ?
```

DB2 V9.7 推出了语句集中器功能，能自动分析动态 SQL 语句，对于只是常量不同的动态 SQL 语句，能让它们自动地共享含有参数标记的 SQL 语句，从而避免修改应用程序。于是，这些相似的 SQL 语句都能命中高速缓存，共享访问计划，节省了编译时间和 CPU 开销。

缺省情况下，语句集中器处于禁用状态。如果希望对所有动态语句启用语句集中器，只需将 STMT_CONC 这个数据库配置参数设置为 LITERALS，如下所示：

```
update db cfg using STMT_CONC LITERALS
```

（2）对于 OLAP 应用，尽可能追求最佳的访问计划。

在 OLAP 应用中，一条复杂查询语句总的执行时间往往较长，编译时间几乎可以忽略不计，所以对这类应用来说，根本目标是获得最好的访问计划。这个时候，对复杂查询使用类似上面的参数标记，得到的访问计划可能不是最优的，这是因为优化器对谓词中参数和常量的基数估计方法是不一样。下面，举例说明。

- where sales.nation = ? ：不论实际执行这个 SQL 语句时传入的参数是什么，优化器将按照下面的公式来计算这个谓词的过滤因子，这可能产生不精确的估计：

```
Filter Factor = 1/ card(nation)
```

- where sales.nation = 'CHINA'：优化器在计算这个谓词的过滤因子时，能充分利用这个字段上的统计信息，从而进行更精确的估计，从而产生更好的访问计划。

总之，在 OLAP 应用中，例如报表或者分析应用，在 SQL 语句中应尽量直接使用常量而不是参数标记，这样才会产生更好的访问计划，查询性能才会高。

- 对于存储过程和嵌入式 SQL 应用，灵活运用 REOPT 功能。

对于存储过程和嵌入式 SQL 应用而言，大量的 SQL 语句都是使用参数或者宿主变量，而这些 SQL 的访问计划在绑定时就生成。如何让这些 SQL 使用最新的统计信息呢？如何根据参数的实际值生成最优的访问计划呢？

这可以通过 REOPT 提供的功能来实现，具体来说，REOPT 有两种工作模式：

- REOPT ALWAYS：对于 SQL 语句的每次执行，都使用变量的实际输入值重新编译。例如，对于 DB2INST1.MYPROC 存储过程，可以使用下面的语句重新绑定程序包，并启用 REOPT ALWAYS 工作模式，如下所示：

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'DB2INST1.MYPROC', 'REOPT ALWAYS')
```

- REOPT ONCE：与 REOPT ALWAYS 会造成过多的语句编译开销不同，REOPT ONCE 仅在第一次执行时，使用变量的实际输入值进行编译和优化，后续的执行会重用执行计划，不再重新编译。当然，第一次执行时输入的值应具有代表性，REOPT ONCE 才能达到优化目的。例如，对于 DB2INST1.MYPROC 存储过程，可以使用下面的语句重新绑定程序包，并启用 REOPT ONCE 工作模式，如下所示：

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'DB2INST1.MYPROC', 'REOPT ONCE')
```

注意：不同的场景采用不同的技术

参数标记、常量、语句集中器和 REOPT，它们的应用场景不同，在实际应用中，应根据场景合理运用。

9.5.10 分组拼接字符串：LISTAGG

很多用户曾为拼接不同行中的字符串烦恼，用递归或者存储过程可以实现这个功能，但是比较麻烦。从 DB2 V9.7.4 开始，新增的 LISTAGG 聚集函数可以非常简单地实现这个功能。

下面使用 LISTAGG 的例子，按照 GRP 字段分组，随后拼接 NAME 字段中不同行的字符串：

```
create table t(grp int, name varchar(5));
DB20000I  The SQL command completed successfully.

insert into t values (1, 'Jim'), (1,'Tom'), (1,'Jane'),(2,'Tony'),(2,'Mike');
DB20000I  The SQL command completed successfully.
select grp,name from t;

GRP         NAME
----------- -----
          1 Jim
          1 Tom
          1 Jane
          2 Tony
          2 Mike

  5 record(s) selected.

--用 LISTAGG 按 GROUP BY 分组拼接字符，组内按名字排序。
SELECT grp, LISTAGG(name, ',') WITHIN GROUP(ORDER BY name) as names
   FROM t GROUP BY grp;

GRP         NAMES
----------  ---------------------
          1 Jane,Jim,Tom
          2 Mike,Tony

2 record(s) selected.
```

注意：LISTAGG 分组的组内序

在使用 LISTGAGG 函数的时候，可以使用 WITH GROUP ORDER BY 子句对分组内的字段进行排序。如本例中，在每个分组内，按 NAME 排序后将字符串拼接在一起。

9.5.11 事务结束后及时 COMMIT

在事务编程中，COMMIT 语句用于提交事务，及时执行 COMMIT 会带来 3 点好处：

（1）COMMIT 后，对数据库所做的更改才会真正落实到数据库，从而避免系统故障导致的数据丢失。

（2）COMMIT 之后，DB2 会释放当前事务在修改的数据（包括 INSERT、UPDATE 和 DELETE）上持有的锁，这些锁可能阻塞了其他应用程序，所以及时 COMMIT 能提高数据库的并发性能。另外，锁也是公共资源，及时 COMMIT 释放的锁资源能供其他事务使用。

（3）COMMIT 之后，当前事务（特别是长事务）占用的日志资源将被释放，可以供其他事务使用。

最后，关于存储过程相关的高效编程原则，请参考《从 Oracle 到 DB2 开发—从容转身》相关内容。

9.6　小结

SQL 语句优化涉及很多方面，最重要的是理解优化器原理，并学会分析 SQL 语句的访问计划。本章首先介绍了优化器的基本原理，深入讲解了访问计划各方面的内容。然后分享了 SQL 调优的实战案例集，这些实际案例融合了优化器、数据库配置、访问计划等各种知识。最后，讲解了高级 SQL 应用和编写高效 SQL 语句的最佳实践。

掌握 SQL 优化技能，不可一蹴而就，要先理解基本概念和原理，然后通过不断实践，最后才能融会贯通。

第10章 DB2 数据仓库设计与优化

本着尊重大师、尊重原创的精神，原文引用数据仓库大师 William H．Inmon 在《构建数据仓库》（Building the Data Warehouse）一书中对数据仓库的定义：

Data Warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision making process.

上面的话的中文意思就是：数据仓库就是面向主题的（subject-oriented）、集成的（integrated）、反映历史变化（time-variant）的、相对稳定（nonvolatile）的数据集合，它可用以支持经营管理中的决策制定过程。

本章将重点讲述 DB2 数据仓库的设计、搭建和优化方法。

10.1　数据仓库是什么

什么是数据仓库呢？下面是笔者对数据仓库大师 William H. Inmon 阐述的数据仓库定义的理解：

- 面向主题：是指数据仓库中的数据是按照一定的主题域进行组织的。
- 集成：是指数据仓库中的数据是在对原有分散的数据库数据抽取、清理的基础上经过系统加工、汇总和整理得到的，是关于整个企业的一致的全局信息。
- 反映历史变化：是指数据仓库中的数据系统记录了企业从过去某一时刻（如开始应用数据仓库的时刻）到目前的各个阶段的信息。
- 相对稳定：是指一旦数据进入数据仓库以后，一般情况下将被长期保留，也就是数据仓库中一般有大量的查询操作，但修改和删除操作很少，通常只需要定期地加载、刷新。

那么数据仓库出现的现实依据是什么？这需要从传统的数据库技术和数据处理模式谈起。

传统的数据库技术往往是以单一的数据源（以数据库为中心）进行事务处理、批处理、决策分析等各种数据处理工作。

数据处理模式主要划分为两大类：操作型处理和分析型处理。操作型处理也叫事务型处理，是指对数据库的日常操作，它通常是对一条或一组记录的查询或修改，满足了响应时间、数据的安全性和完整性的需要。分析型处理则用于管理人员的决策分析，往往是大规模的、批量的计算作业，经常要访问大量的历史数据。

由此可以看到，传统数据库系统能够完成企业的日常事务处理工作，但很难实现数据分析处理的要求，也无法满足数据处理多样化的需求。随着用户需求的发展，操作型处理和分析型处理的分离就成为一种必然选择。于是，专门用于分析型处理的系统，即数据仓库，便应运而生。

10.1.1　数据仓库的误区

误区 1：数据仓库是一个产品，买到就能使用。

目前，一流的软件厂商都会提供完整的数据仓库解决方案，以数据流的角度来看，从对源数据 ETL 的产品，到支撑数据仓库存储的数据存储产品，再到业务模型产品，最后到用户直接看到的展现层面的产品。完善的数据仓库解决方案可以帮助用户在构建数据仓库的各个阶段，轻松便捷地完成相应的工作。

数据仓库是以关系数据库并行处理与分布式处理、以及联机分析处理等技术的发展为基础，为解决当前企业和组织中虽然拥有大量数据、但信息贫乏的现状而提出的，是一种对不同系统数据实现集成和共享的综合性解决方案。所以，数据仓库不能简单地理解成一个产品，它是一套架构、一套解决方案。数据仓库必须经历构建的过程，根据用户实际业务需求、现有业务系统信息情况进行构建的过程。

误区 2：数据仓库是一个项目，有开始和也有结束。

有一个朋友对我说，“我正在为某单位做一个数据仓库项目，预计 2 个月后结束。”从他

的角度来说，这种说法貌似也没错。但实际上，对于数据仓库来说，它是一个过程，数据清洗、整合、计算和提供展现的过程，它是持续的，更像是一段没有终点的旅程。

在这样的旅程中，对 DBA 的要求和责任也是很高的。面对海量数据（TB 级，PB 级）的数据仓库，怎样保证日常数据加载的效率，又不影响服务？怎样提供高可用性的仓库？怎样实现容灾？这一系列问题是 DBA 必须面对和解决的。

误区 3：数据仓库和数据库产品没区别，可以直接使用一般数据库做为数据仓库。

有朋友曾经问我，“数据仓库不就数据存的多些，我在数据库中多定义一些表，多存点数据不就行了？！在普通的操作型数据库（OLTP）中就能够实现，对吧？”

这是由于不清楚操作型数据库（OLTP）和数据仓库（OLAP）系统各自的特点，才会有这种想法。

实际上，数据仓库和操作型数据库既有区别，也有联系。数据仓库的建立并不是要取代操作型数据库，相反，它依赖于操作型数据库，并以此为基础，建立一个综合的和完善的信息分析系统，用于支持各级管理层决策分析。

数据仓库弥补了原有数据库的不足，将原来的以单一数据库为中心的数据环境发展为一种新的体系环境。它具有一种新的数据处理结构体系，能够将不同环境、不同系统的数据统一起来，形成综合的中央数据仓库。

数据仓库和普通的操作型数据库不同，它是面向主题（以主题为导向）的，支持商务决策而不是事务处理。它拥有许多优化设计的层次和结构化的查询功能，并以汇总/综合系统为构架。基于对数据快速和有效的分析，数据仓库可为决策系统提供强有力的支持。在开发人员和用户的协同配合和精心设计下，它能够实现对数据的一系列转化，包括从数据到信息，从信息到知识，最终到商业智能。

从 DBA 的视角出发，对操作型数据库和数据仓库系统的主要区别做个对比，如表 10-1 所示：

表 10-1　数据仓库系统与操作型数据库特点比较

比较内容	数据仓库系统	操作型数据库
目标	OLAP 联机分析处理	OLTP 联机事务处理
作用	面向主题	面向过程
活动特征	分析式	事务处理
构成	集成的	异构的、分散的
内容更改频率	不频繁更改	频繁更改
时间性	时序性、历史性	当前的
基础结构	多维型	关系型
关系结构	星型/雪花型结构或混杂型结构	3NF 三级范式
终端用户	多为管理人员和决策人员	多为操作人员

其中，雪花型结构是在星型结构上的进一步扩展。混杂型结构（Hybrid）兼顾星型结构和关系型的特点，具有某些杂交优势，它往往是根据实际应用情况实施结构优化设计的结果。

10.1.2　数据仓库的体系结构

作为 DBA，如果不了解数据仓库的体系结构，不了解数据仓库的特点，就无法胜任数据仓库的运维工作。

如图 10-1 所示，数据仓库的数据源来自操作型数据库，即业务数据系统。在抽取、清理、装载（ETL）处理系统的控制下，数据要经历“艰难”的历程，按照星型、雪花型、混杂型等模型存储在数据仓库系统中，形成一个新的集成系统用于联机分析（OLAP）。最终，通过 OLAP 服务器，向用户提供数据分析、数据报表以及数据挖掘等服务。

其中，ETL 过程、数据仓库的检测和维护是数据仓库系统的重中之重，同样也是 DBA 的工作重点和难点。

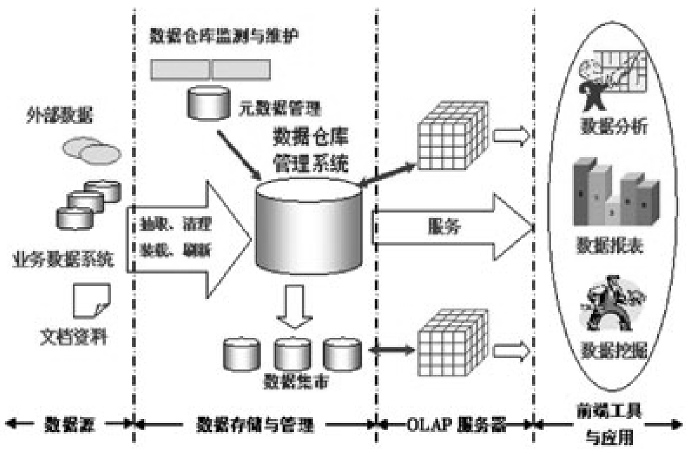


图 10-1　数据仓库的体系架构

那么如何提高数据加载的效率？如何保证数据仓库的性能及高可用性？面对 TB 级、甚至 PB 级数据，数据仓库怎么进行灾备？

上述这些问题不仅涵盖数据仓库的设计，也包含运维的内容，作为 DBA，必须面对、思考这些问题。下面来看看 DB2 提供了哪些特性来支持大容量数据仓库。

10.2　支撑数据仓库的 DB2 特性

数据仓库系统具有数据量大、查询复杂、对响应时间要求高等特点，为了满足这一要求，DB2 提供了针对数据仓库的特性。这里着重介绍 DB2 提供的分区特性、针对分区的连接（JOIN）策略以及物化查询表等特性。

10.2.1　分区数据库

分区数据库（Partitioned Database）技术是指将数据分散到不同的数据库分区中的技术。这里所说的数据库分区是数据库的一部分，它包含数据库中的部分数据，并具有独立的索引、配置文件和事务日志。分区数据库在单分区环境和多分区环境下特点不同，接下来进行具体介绍。

1. 单分区环境

在这种环境下，数据库只有一个分区，它支持单个 CPU 和多个 CPU 的硬件环境，这里只谈谈多 CPU 硬件环境，即 SMP（Symmetric Multiprocessor System）系统。

典型的 SMP 环境是在一台具有多 CPU 的服务器上创建的唯一数据库分区，如图 10-2 所示。

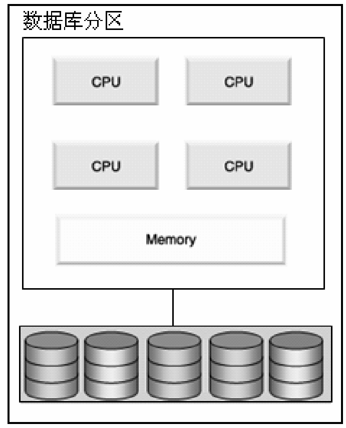


图 10-2　SMP 架构示意图

在SMP环境下可以实现分区内并行。由于具有多个CPU，所以不同的数据库操作都可以快速完成。对于查询任务，DB2 数据库系统将查询分配给不同的 CPU，来提高处理速度。对于其他的数据库操作，比如数据加载、备份与恢复表空间，以及创建索引，都会因多 CPU 取得很好的处理速度。因为数据加载命令、备份与恢复命令都有相应的参数指定并行度，能充分利用多 CPU 从而提高处理速度，具体内容请参考本书“数据库备份与恢复”章节的内容；DB2 在 SMP 环境下支持并行索引维护，可以充分利用多 CPU 加快索引的创建。

在扩展性方面，由于已有了多个 CPU，I/O 方面可能存在瓶颈，不过可以通过增加磁盘驱动器的个数，来提升与 CPU 对应的数据库分区的 I/O 性能，也可以使用专门的 I/O 服务器来专门处理 I/O 请求。

最后，如果目前的 SMP 系统已经达到了性能或扩展的极限，可以考虑将系统迁移到多分区数据库上。

2. 多分区环境

一个数据库由多个数据库分区组成，数据库分区位于不同的服务器上。多个服务器和多个数据库分区组合在一起，形成一个完整的数据库。多分区数据库支持两种硬件环境。

（1）MPP（Massively Parallel Processing）

每个数据库分区都有自己独立的服务器环境，即独立的 CPU、内存以及磁盘。服务器之间通过高速网络连接，这种环境也被叫做 MPP 环境，或者 Shared-Nothing 架构，如图 10-3 所示。Shared-Nothing 这个名字准确地描述了在这个环境中资源的管理方式，与 SMP 环境不同，MPP 环境是由相对独立的、不共享的数据库分区组成。

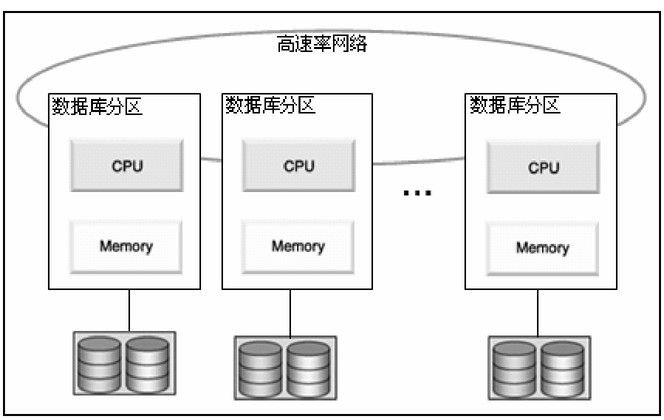


图 10-3　MPP 架构示意图

在这种环境中，尽管一个数据库被物理上分割成多个数据库分区，但从逻辑上看它依然是一个整体。当处理查询时，协调节点（Coordinator Node）将工作负载分配到各个分区上，这些分区基于自己管理的数据进行并行处理，随后按照工作要求，将执行结果返回给协调节点，协调节点合并所有结果，再返回给应用。

（2）SMP Cluster

对于前面讲的 MPP 环境，还有另外的配置方法，可以对每个数据库分区配置多个 CPU 来提升处理能力，这种配置被称作 SMP Cluster。如图 10-4 所示。

那么这种配置方法有什么优点呢？这种配置结合了 SMP 和 MPP 的优点，意味着一个查询语句可以在多个分区中并行处理，也意味着一个查询语句可以被一个分区中的多 CPU 同时执行。

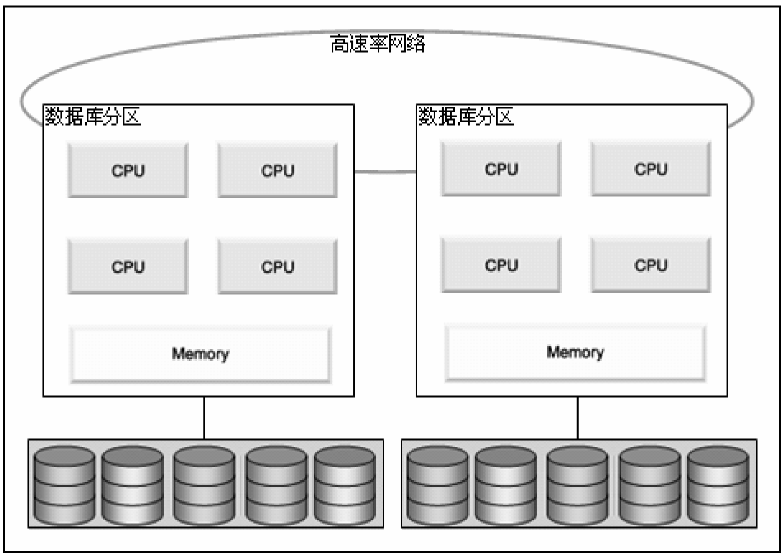


图 10-4　SMP 集群架构示意图

10.2.2　灵活的数据分区

这一节将介绍多分区数据库技术在 DB2 中的具体实现，即 DB2 的数据库分区特性（Data Partitioning Feature），也简称为 DPF。另外，DB2 还提供了表分区（Table Partitioning）和多维聚簇（Multidimensional Clustering）特性，本节也一并介绍。由于 DPF、表分区和 MDC 已在第 5 章做了详细介绍，本节只就这些技术在数据仓库中的使用要点进行讲解。

1. DPF 数据库分区

DPF 采用的是 Share-Nothing 体系结构，依托于构建在 MPP 或 SMP Cluster 环境上的多分区数据库，实现了对数据操作的并行（分区间的并行），从而提升了数据库系统的 I/O 能力和响应速度，架构如图 10-5 所示。

在 DPF 中创建数据表时，需要在建表语句（CREATE TABLE）的 DISTRIBUTE BY 子句中指定分区键。以下面的建表语句为例，DB2 数据库管理器会根据表中 user_id 列进行哈希计算，从而将该表的数据分布在不同的数据库分区中，这里的 user_id 被称作分区键。

```
CREATE TABLE user_info( user_id INT, user_name CHAR(8)) DISTRIBUTE BY(user_id)
```

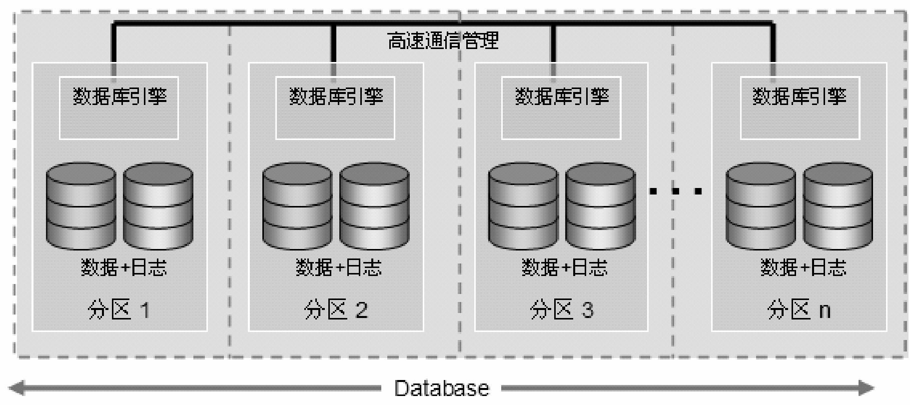


图 10-5　DPF 架构示意图

注意：表中的数据会分布在所有的数据库分区中吗？

这个问题的答案涉及到以下几个知识点，下面分别介绍一下。

第一个知识点：数据库分区组。

分区组包含部分或全部数据库分区的组合。在 DB2 中，创建数据库分区组的命令如下:

```
CREATE DATABASE PARTITION GROUP 数据库分区组名
   ON DBPARTITIONNUM (数据库分区号 1,数据库分区号 2, ... 数据库分区号 n)
```

或

```
CREATE DATABASE PARTITION GROUP 分区组名
   ON DBPARTITIONNUM (数据库分区号 1 TO 数据库分区号 n)
```

第二个知识点：数据库表空间。

在多分区数据库环境中，创建表空间需要指定它所在的数据库分区组，如果不指定分区组的名字，数据库系统将使用默认的数据库分组 IBMDEFAULTGROUP。创建表空间的命令如下:

```
CREATE TABLESPACE 表空间名 IN DATABASE PARTITION GROUP 数据库分区组名
```

在 DB2 数据库中，表被创建在表空间中。因此，表空间对应的数据库分区组，决定了创建在该表空间上的数据表对应哪些数据库分区。也就是说，表中的数据可能会分布到的数据库分区，是在创建表空间时就已经设定好的。

需要注意的是，分区键选择的好坏会对分区数据库的性能产生重大影响。如果数据不能相对均衡地分布在所有数据库分区中，而是部分数据库分区有大量数据，而其余分区只有少量数据时，数据处理的工作负载就会集中在这些拥有大量数据的数据库分区上，而其余的数据库分区则基本上处于空闲状态。这会导致系统资源的极大浪费，使数据库无法达到它应有的性能。分区键的选择原则如下：

首先，对分区键的操作效率而言，最优的数据类型是整数类型，其次是字符型，最后是小数型。对于所有的长数据类型（比如 LONG VARCHAR、LONG VARGRAPHIC）和大对象数据类型（比如 BLOB、CLOB、XML）是不能被作为分区键使用的。

其次，从查询效率的角度来说，经常在连接（Join）中使用的列是最佳的选择。因为这样能充分利用分区数据库的特性，使数据库可以直接在每个分区中完成表的连接（具体内容在下节中有详细阐述）。

再次，最佳的分区键选择是要实现数据均匀地分布在分区中，这样能够将工作负载均摊到每个数据库分区上，从而从整体上提升数据库的读写效率。

最后，切记，一定不能使用经常被更新的列作为分区键。因为分区键的值被更新时，会引起数据在分区中的重新分布，将导致系统的额外开销，从而影响数据库的性能。

2. 表分区

表分区是将表中的数据按照特定的列进行划分，表存储结构会根据所选定的列进行排列以优化表的读写，同时不同的表分区可以存放在不同或者相同的表空间中。这是一种在表的层面进行数据分区的技术。

创建表分区的语法是，使用带有 PARTITION BY RANGE 子句的建表语句（CREATE TABLE）。例如下面的语句，创建了表 customer，并以列 shipdate 作为表分区键来对表进行分区。其中在“01/01/2011”和“03/31/2011”之间的数据会被存储在表空间 ts1 中，在“04/01/2011”和“06/30/2011”之间的数据会被存储在表空间 ts2 中，以此类推。

```
CREATE TABLE customer (shipdate DATE, name CHAR(30))
IN ts1, ts2, ts3, ts4
PARTITION BY RANGE(shipdate) (STARTING FROM ('01/01/2011')
ENDING AT ('12/31/2011') EVERY (3 MONTHS))
```

使用表分区有以下好处：

- 提高查询效率。DB2 优化器将限定 SQL 执行时，只访问 WHERE 语句中涉及的数据分区，避免访问无关数据分区，从而提高了查询效率。
- 可根据需要轻松地滚入（Roll-in）和滚出（Roll-out）数据分区。
- 数据的压缩率不会受到新数据的影响。这是因为每一个表分区都会有自己的压缩字典。当有新数据插入时，不会影响已有数据分区的压缩率。

3. MDC

MDC 是一种可以按多个维度把数据按照块进行存储的技术，它可以把在维度上有类似值的行聚集在一起存放在物理连续的块中。当 WHERE、GROUP BY 和 ORDER BY 等子句中包含有 MDC 表中定义的维度列时，查询语句的执行效率可以得到显著的提升。

可以使用带有 ORGANIZE BY 子句的建表语句（CREATE TABLE）来创建 MDC 表。例如下面的语句创建了一个名为 mdc_tbl 的 MDC 表，ORGANIZE BY DIMENSIONS 子句指出使用列 region、store 和 year_and_month 作为维度。

```
CREATE TABLE mdc_tbl (
date DATE,
store CHAR(8),
region CHAR(2),
year_and_month generated as INTEGER(date)/100
```

```
)
ORGANIZE BY DIMENSIONS (region, store, year_and_month)
```

维度选择的是否合适将会对 MDC 表的性能产生重大影响。合适的维度会显著地提升访问 MDC 表的性能，而不恰当的维度会使 MDC 表浪费大量存储空间，并带来极差的效率。在维度的选择上，要遵循下面的原则：

第一，可以选择经常在谓词中使用的列。如上例，在商品销售表（mdc_tbl）中定义了商品信息（store）、销售时间（year_and_month）、销售区域（region）等列。如果使用销售区域（region）和销售时间（year_and_month）为 WHERE 条件进行 SELECT 查询，那么由于这两列是 MDC 维度，所以在此 WHERE 条件上的查询速度要远远快于非 MDC 方式存储的表。

第二，要选择具有粗粒度的列，不能选择细粒度的列。比如，如果一个时间列中的值精确到了日（例如，2011-10-01），那么这列就不适合作为 MDC 表的维度。由于这种细粒度的列会导致系统分配大量的利用率极低的存储块来存储 MDC 表。它既浪费了硬盘资源，也降低了系统性能。如果时间列中的值精确到月或年（例如，2011-10 或 2011），这样的列就可以被作为 MDC 表的维度。

第三，查询语句中，被 GROUP BY 或者 ORDER BY 子句使用的列，可以被做为 MDC 的维度使用，这样能够提高查询效率。

最后，一定要注意，维度的选择最好不要超过三个，否则会降低系统性能。

10.2.3 多分区环境下的连接（Join）策略

数据仓库系统经常需要对多个大表进行连接操作，这就需要数据库能够提供高效的查询处理能力。DB2 提供的分区技术可以将数据在多个分区中平均分布，从而提高了查询的并行处理能力，降低了查询的整体执行时间。

但是，在多分区环境下，连接（Join）应当采用什么样的方式和使用什么样的策略呢？本节将讲述 DB2 优化器如何高效地处理多分区环境下的连接操作。

注意，在下面的讲解中统一使用两个表 CUST 和 SALES。其中，CUST 是客户信息表，表中定义了客户编号（CUST_ID）列；SALES 是销售记录表，表中也定义了用户编号（CUST_ID）列，与客户信息表相关联，并使用同样的查询语句。

```
select * from cust c,sales s where c.cust_id=s.cust_id
```

还有一点需要注意的是，在以下图例中，表 CUST 中 CUST_ID 列的值和 SALES 表中 CUST_ID 的值，分别用圆圈数字和方块数字表示。例如，CUST 表中有一条记录的 CUST_ID 值为 1，则用圆圈 1 表示；SALES 表中有一条记录的 CUST_ID 值为 1，则用方块 1 表示。

1. 并行连接（Collocated joins）

如图 10-6 所示，表 CUST 和表 SALES 使用同样的列（CUST_ID）作为分区键，并且使用同样的数据分区图（Distribution Map）。这两个表中的数据拥有同样的数据分布，在同一个分区中可以对表中的数据进行直接关联。

当 DB2 在处理查询语句时，会采用并行连接的策略。即，在两个分区中分别对两个表进行连接，在分区间不会有数据的移动，最终的查询结果由两个分区的连接结果合并组成。

2. 直接连接（Directed joins）

如图 10-7 所示，表 CUST 使用 CUST_ID 列作为分区键，表 SALES 使用其他不同的列作为分区键。

在这种情况下，DB2 在处理上述查询语句时会采用直接连接。即 DB2 会对表 SALES 中的列 CUST_ID 进行哈希，并使用表 CUST 的分区图将表 SALES 中的数据行直接发送到表 CUST 所在的数据库分区上，表 CUST 与表 SALES 的连接将在相应的分区上进行。

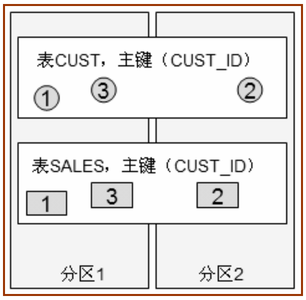


图 10-6　采用并行连接的数据分布示意图

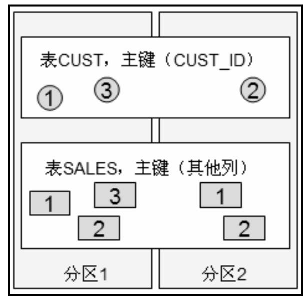


图 10-7　采用直接连接的数据分布示意图

3. 广播连接（Broadcast joins）

如图 10-8 所示，表 CUST 不使用 CUST_ID 列为分区键，而使用其他的列；表 SALES 使用 CUST_ID 列作为分区键。其中，表 CUST 中的数据比较少。

通常情况下，当查询语句中不包含相等连接的谓词时，DB2 会选择使用广播连接的策略。但是，如果查询语句中包含相等连接的谓词，DB2 认为执行查询语句使用广播连接比使用重分区连接（Repartitioned joins）的开销要少时，DB2 也会选择使用广播连接。

在采用广播连接策略时，DB2 先扫描表 CUST，分析连接谓词，将 CUST 表中的所有数据发送到表 SALES 所在的所有数据库分区上。然后在各个分区上，DB2 对表 SALES 进行扫描，分析谓词，与发送过来的表 CUST 中的数据进行连接。最终各分区将连接结果返回。

4. 重分区连接（Repartitioned joins）

如图 10-9 所示，表 CUST 和表 SALES 中的分区键都不是连接谓词中使用的列 CUST_ID，这时 DB2 采用重分区连接策略，即根据连接谓词中的列（CUST_ID）把表 CUST 和表 SALES 进行哈希运算，然后把两个表中的数据发送到哈希运算后对应的数据库分区上，在这些数据库分区上把表 CUST 和表 SALES 进行连接，最终各分区把连接结果返回。

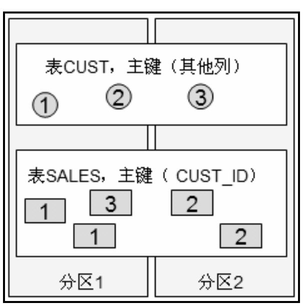


图 10-8　采用广播连接的数据分布示意图

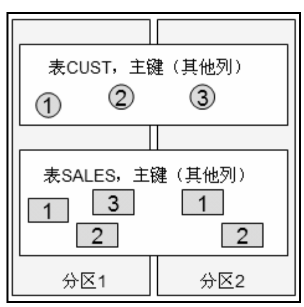


图 10-9　采用重分区连接的数据分布示意图

10.2.4 物化查询表

物化查询表（MQT）和分区特性一样，都是为提升数据仓库应用的性能而设计的。MQT的创建是基于一个查询结果，它里面存放的是经过预计算得到的结果。

创建 MQT 的命令如下，物化查询表 sales_prod 是由两个表 product 和 sales 关联生成的，其中表 product 是产品表，包含字段 prod_id 和 prod_desc，表 sales 是销售信息表，包含字段 prod_id 和 amount。MQT 创建后，可以使用“REFRESH TABLE”命令更新表 sales_prod 的内容。

```
CREATE TABLE sales_prod AS
(
SELECT p.prod_id,
        p.prod_desc,
        s.amount
 FROM product p, sales s
 WHERE p.prod_id = s.prod_id
)
DATA INITIALLY DEFERRED REFRESH DEFERRED;

REFRESH TABLE sales_prod;
```

在数据仓库系统中，MQT 对下面几类复杂查询具有很好的性能：

- 在表中对数据的聚类或计算。
- 表之间的连接。
- 一个较大表中的子集数据被频繁访问。

关于 MQT 的具体讲解、使用方法和设计原则，请参考第 5 章有关内容。

10.3 DB2 数据仓库设计

最近去一家人寿保险公司进行数据仓库技术服务，发现他们的数据仓库系统在海量数据处理方面没有进行过合理地设计，没有发挥出并行处理的优势；此外还注意到，当前 DBA 参与设计的比例越来越大了。因此，在这一节笔者将分享数据仓库设计阶段的一些经验。

10.3.1 BCU/BPU 的设计原则

在介绍 BCU/BPU 的设计原则之前，首先要弄明白什么是 BPU，什么是 BCU。

1. BPU（Balanced Partition Unit）

BPU 是一个逻辑概念，是指服务于一个 DB2 数据库分区的一组相关功能和相关系统资源。

如图 10-10 所示，一个 BPU 被定义为一个 DB2 数据分区和相关的一些资源，包括 DB2 数据分区、DB2 Agent、为 Agent 服务的 CPU 和内存，以及存放分区数据的存储。

根据服务的不同，可以将 BPU 划分为以下三种类型：

- **Data BPU**：用于存放用户数据的数据库分区；

● **Coordinator BPU**：用于管理用户连接，负责协调查询的数据库分区；

● **Catalog BPU**：用于存放DB2编目的数据库分区。

图 10-10　BPU 示意图

在进行数据仓库的容量规划时，BPU 非常有用。比如：假定把一个 BPU（一个 DB2 的数据库分区）配置为可以支持 250G 的裸数据，假定数据仓库总的规划容量为 8TB 裸数据，那么整个数据仓库将需要 32 个 BPU。利用这个数据可以推算出相应的 BCU（如果一个 BCU 上支持 8 个 BPU，即需要 4 个 BCU）。由于 DB2 在不同的数据库分区之间平均分配数据和资源，所以一旦从 CPU、内存、I/O 等角度确定了一个 BPU 的系统需求，就可以确定整个数据仓库的硬件配置。

在系统设计和实施阶段，BPU 也非常有用。可以在 BPU 层次设计物理硬件、存储、数据库参数配置等，然后把该 BPU 的配置映射到整个 BCU。这样就大大简化了系统的初始设计、配置、管理、维护等工作。

2. BCU（Balanced Configuration Unit）

BCU 是指把 CPU、内存、存储、分区、配置参数等结合在一起的、可扩展的基本单元。BCU 是最小的、可复制的软硬件组合。针对数据仓库系统，BCU 保证了基础构架上的扩展性，提供了磁盘 I/O、内存、CPU、网络的优良扩展性能。

BPU 与 BCU 的关系，如图 10-11 所示：

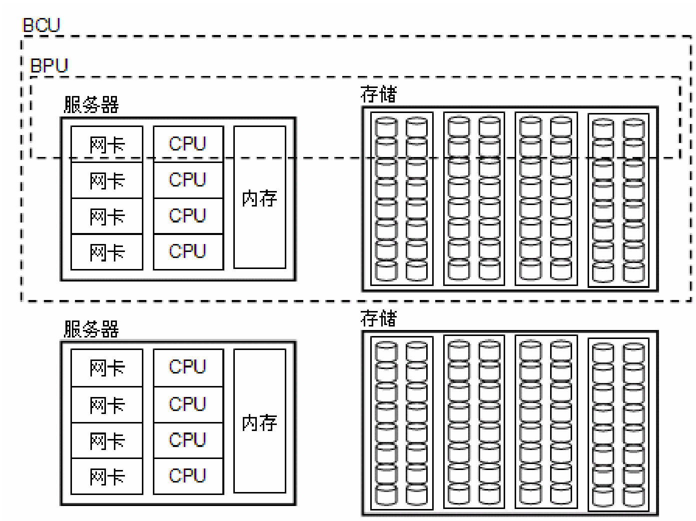


图 10-11　BPU 与 BCU 的关系示意图

BCU 的类型主要分为以下两种：

● Data BCU：是 DB2 数据库分区的集合，用于管理和处理数据库中的用户数据。一个 BCU 支持一个或多个 BPU。

- Admin BCU：是 DB2 数据库分区的集合，作为 DB2 的编目分区（Catalog Partition）和协调节点（Coordinator Node）。

比较：编目分区和协调节点有什么区别？

编目分区（Catalog Partition）

在创建数据库的时候，数据库管理系统会自动创建几个表空间，其中一个是 SYSCATSPACE，它用来保存 DB2 的编目信息（Catalog）。在分区数据库系统中，SYSCATSPACE 表空间只保存在一个分区上，这个分区就是编目分区（Catalog Partition）。

也就是说，在哪个分区上运行了创建数据库命令（CREATE DATABASE），对于这个数据库而言，该分区就是编目分区，SYSCATSPACE 表空间将创建在该分区上。

如果想弄清楚哪个分区是它的编目分区，可以使用命令“list db directory”。在这个命令的返回信息中，有一项是编目数据库分区号（Catalog Database Partition Number），它指出了当前数据库的编目分区。当然，也可以通过从不同分区上连接数据库，然后使用命令“list tablespaces”，显示 SYSCATSPACE 表空间的分区就是编目分区。

协调节点（Coordinator Node）

协调节点是运行着协调代理（Coordinator Agent）的分区，任意一个分区都有可能成为协调节点。

在 DB2 中，协调代理（Coordinator Agent）与应用进行交互，负责接收请求和返回结果，它可以自己处理应用的请求或者将工作布置给多个副代理（Subagent）处理。

协调节点可以通过命令 SET CLIENT CONNECT_NODE 来指定，当应用发出一个查询要求，协调节点将查询请求发送给其他分区上的副代理，各个分区上的副代理将各自的结果返回给协调节点，由协调节点把查询结果聚合后返回给应用。

在 BCU 的设计中，一个最重要的设计原则是分离管理器 BPU 和数据 BPU，让它们独立承担各自的工作。典型的 BCU DB2 数据库设计，如图 10-12 所示，该示例中，定义了一个管理 BCU（包含一个管理 BPU），两个数据 BCU（各包含 8 个数据 BPU）。

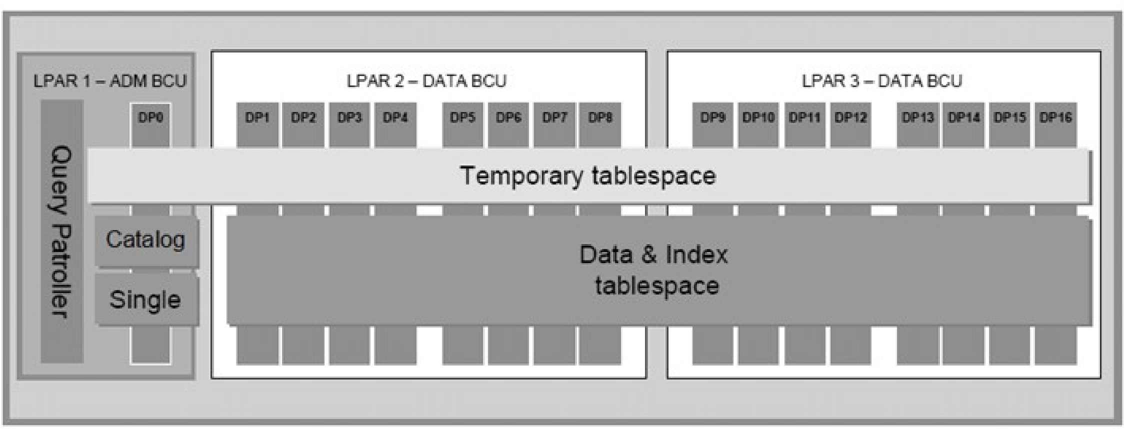


图 10-12　BCU 的设计示意图

管理器 BPU 是一个 DB2 数据库分区，它主要有以下三个角色：

- 保存数据库的编目表；
- 扮演协调节点，响应用户连接；
- 存储小型表和没有分区的表。

数据 BPU 同样也是 DB2 数据库分区，它的主要职责是负责存储被分区的表数据。

10.3.2　数据 BPU 上存储划分

每一个数据 BPU 都有自己的数据库引擎、数据库内存、数据库日志和锁，并且要独立地操作一组数据子集。所有的数据 BPU 应该拥有同等数量的数据。这样会保证每一个数据库分区在处理查询时具有同样的工作负载，也就是在处理一条查询时，所有的数据库分区都同时进行工作。

从 CPU 和内存的角度来看，数据 BPU 是一个逻辑实体，因为数据 BPU 所使用的内存和线程，并非归这个数据 BPU 固定占有和使用，而是由操作系统动态分配。也就是说，在一台服务器上的任意一个 CPU，会为在这台服务器上的任意数据库进程服务，因此 CPU、内存等系统资源独立于运行该进程的数据库分区。

关于存储的设计，原则是保证每一个数据 BPU 都有属于自己的一组物理磁盘，并且保证数据仓库中的所有数据对象都均匀地分布在这些磁盘上。这种设计模式，可以保证每个 BPU 中的所有数据对象，在 BPU 中都会均等地进行 I/O 操作。

10.3.3　数据库的文件系统划分

关于数据库使用的文件系统的设计原则，对于 BPU，建议创建不同的文件系统用来存放数据库中不同的信息，例如，设计四个独立的文件系统（db2path，db2fs，db2plog，db2mlog）分别存放数据库实例、表空间、主日志以及镜像日志。在每个文件系统下，根据实例名（instance name）创建属于不同 BPU 的文件系统。

1. /db2path/instancename/NODE000*N*

这个文件系统是 BPU N 的数据库目录。注意，如果 N=10，文件系统的命名为/db2path/instancename/NODE0010。

2. /db2fs/instancename/NODE000N

这个文件系统存放 BPU N 上的所有永久的或临时的表空间容器。在 BPU 0 上，它对应的文件系统同时也存放着数据库的编目表。

在每个数据仓库的节点上都会创建命名为/db2fs/instancename/NODE000 *N*/db2dump 的文件系统，用来存放诊断日志文件。同样的软连接/db2fs/instancename/db2dump 作为诊断日志参数 DIAGPATH 的值，设置在所有的管理节点和数据节点上，而这些连接指向每一个节点的诊断日志子目录。

3. /db2plog/instancename/NODE000N

这里保存着 BPU N 的主日志。

4. /db2mlog/instancename/NODE000N

这里存放着 BPU N 的镜像日志。

10.3.4　仓库中诞生的数据库

可以使用以下语句创建数据仓库中的数据库：

```
CREATE DATABASE testdb ON /db2fs DBPATH ON /db2path PAGESIZE 16384
DFT_EXTENT_SZ 16 AUTOCONFIGURE APPLY NONE;
```

注意：testdb 是数据库的名字，执行上述命令的分区就被称作编目分区。

使用这条语句创建数据库时，同时意味着：

（1）开启了数据库的自动存储管理功能。

（2）路径/db2fs 作为所有的自动存储表空间的容器。

（3）路径/db2path 作为数据库的目录。

（4）Buffer pool 和表空间的默认页大小为 16KB。

（5）默认的扩展块大小为 16 个页，也就是说，当页大小为 16KB 时，扩展块大小为 256KB。

10.3.5　数据库分区组的设计

在创建 DB2 数据库的同时，下面这三个默认的数据库分区组会被自动创建。

- IBMCATGROUP：这个数据库分区组用于存放编目表。该分区组只创建在一个数据库分区上，可以指定它保存在 0 号数据库分区上。
- IBMTEMPGROUP：这个分区组在所有的数据库分区上都有，系统管理的默认临时表空间 TEMPSPACE1 就保存在这里。
- IBMDEFAULTGROUP：当数据库创建后，这个分区组是默认的表空间分区组，如果在创建表空间时不指明分区组，就使用这个分区组。这个分区组包含数据库实例中所有的数据库分区。另外，它包含着一个名为 USERSPACE1 的表空间，当用户创建表时，如不指明表空间，都会默认地把表建在这个表空间中。

除了上面默认的分区组外，还会创建用户分区组。分区组的名称可以自行设定，本例中命名为 SDPG 和 PDPG，分别用于管理 BPU 和数据 BPU：

- SDPG：存放单分区数据，并用于管理所有 BPU 的分区组。这个分区组只包含一个数据库分区，即数据库分区 0（BPU 0）。
- PDPG：存放分区数据的分区组。这个分区组只包含数据 BPU，用来保存用户数据。

创建上述分区组的命令如下，假设有 1 个管理 BPU 和 16 个数据 BPU，管理 BPU 是第 1 个数据库分区，即分区号为 0，数据分区包括分区 1~分区 16。

```
CREATE DATABASE PARTITION GROUP SDPG ON DBPARTITIONNUMS (0);
CREATE DATABASE PARTITION GROUP PDPG ON DBPARTITIONNUMS (1 TO 16);
```

数据库分区组的设计情况如图 10-13 所示：

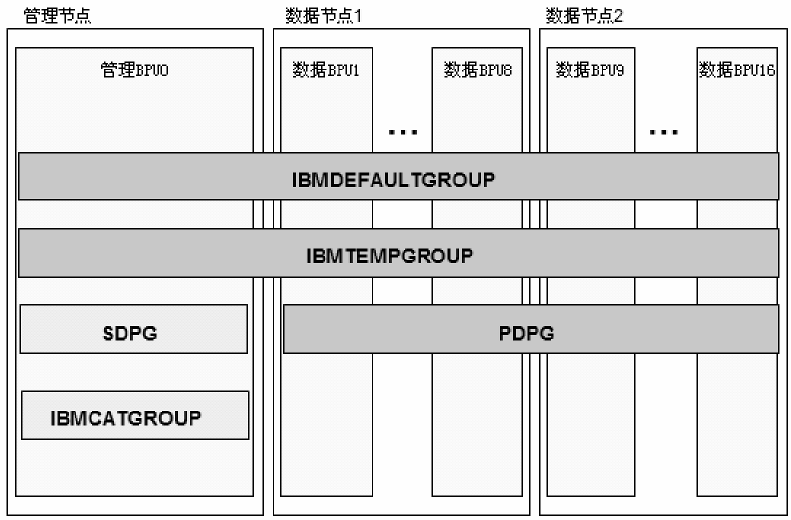


图 10-13　数据库分区组的设计示意图

10.3.6　缓冲池的设计

数据库管理器使用缓冲池对数据进行缓存，对于每一个表空间页面的大小，必须至少有一个相同页大小的缓冲池存在。因此，与表空间一样，缓冲池的页大小也有 4KB、8KB、16KB 和 32KB。

注意：缓冲池的数量不用太多

在数据仓库中，缓冲池的数量不用太多！这是因为数据库管理器对单一缓冲池中的数据、索引和临时页管理起来具有更高的效率。如果一个环境中定义了多个缓冲池，数据库管理必须判断出哪个缓冲池有更多的页，确保每个缓冲池达到了它的最高效率，这就增加了额外的消耗。

维表、字典表通常保存在管理 BPU 上，由于它经常访问，所以缓冲池的命中率会非常高。如果这些类型的数据也要保存在数据节点上，采用一个独立的缓冲池会有更高的效率。

创建基于块的缓冲池也是一个很重要的选择。基于块的缓冲池由块区域和页区域组成。在块区域中有指定数量的连续页，用来处理连续预读的工作；页区域用来处理其他所有的 I/O 工作。推荐保留三分之一的缓冲池用作块区域。

使用基于块的缓冲池，在有预读需要时，通过块 I/O 可以提升顺序读和全表扫描的性能。顺序读时，DB2 的预读机制将磁盘上临近的块读入连续的内存页中，预读机制采用的块 I/O 的效率远远高于默认的基于页的 I/O。

10.3.7　数据库日志的设计

如果数据库分区中的一个或多个表空间需要被重建，可以通过在此分区上的备份介质进行恢复并执行 roll-forward 操作。因此，建议把单个分区的表空间容器和活动日志保存在不同的存储上。

通过设置数据库配置参数 NEWPATHLOG 重新指定日志文件的存放位置，下面的命令将日志的默认位置改为/db2plog/bcuaix：

```
db2 update db cfg for bcudb using NEWLOGPATH /db2plog/bcuaix;
```

需要注意的是，在给参数 NEWPATHLOG 设置了值之后，路径/NODENNNN（其中，NNNN 是分区的编号，例如第 9 号分区就是 NODE0009）会被自动增加到参数 NEWPATHLOG 的值后，这样某个节点的实际日志存放位置为/db2plog/bcuaix/NODENNNN。

注意：关于镜像日志的设计

从数据库的层面上看，镜像日志是在活动日志被意外删除或者外部存储出现数据破损的情况下，保证还有额外的镜像文件作为活动日志，从而确保数据库正常运行。

在设置镜像日志后，每一个日志文件会被同时写两份，如果数据库的主日志文件无法写入时，数据库会标识主日志文件为坏文件，并且继续将日志写入镜像文件。在执行回滚或者前滚操作时，数据库服务器会自己判断哪个日志文件可用。

如下所示，可以通过设定参数 MIRRORLOGPATH 来启用镜像文件。例如，将值/db2mlog/bcuaix 赋给该参数后，路径/NODENNNN 会被自动添加到镜像参数值的后面，这也就是镜像日志在节点上的实际存放位置。

```
db2 update db cfg for bcudb using MIRRORLOGPATH /db2mlog/bcuaix;
```

10.3.8 表空间的设计

对于用户表空间，推荐使用自动存储管理。

可以使用如下命令创建一个横跨所有数据 BPU 的数据库管理类型（DMS）表空间。

```
CREATE TABLESPACE TS_BIG IN DATABASE PARTITION GROUP PDPG BUFFERPOOL BP_16K;
```

这个表空间默认的页大小是 16K，默认的扩展大小为 16 个页，默认的预取大小为 384 页，它使用缓冲池 BP_16K。

注意：默认的表空间是哪个？

在创建表时，设计原则要求在 create table 命令中明确指出使用哪个表空间（Table Space），然而，很多应用在开发时并没有在建表语句中显式地指定所使用的表空间，这种情况下，DB2 会使用默认分区组 IBMDEFAULTGROUP 中的默认表空间 USERSPACE1 来存放用户表。

10.3.9 表的存放技巧

对于数据量不同的表，通常表的存放技巧也各不相同。

对于大表，强烈推荐将这类表存放在跨所有数据 BPU 的独立表空间里，另外也结合使用表分区和 MDC 技术。

对于中型表，这类表可以按逻辑分组，五六个或更多一点的表存放在一个表空间上，这些表空间也要横跨所有数据 BPU。同一个表空间上的表，最好在逻辑上有一定的关系，例如所有

数据都是同一个部门的。这类表的索引，可以存放在与数据相同的表空间中，也可以独立存放。

对于小型表，多数这类表会被放在单一分区的表空间上，通常位于管理 BPU 上。索引和数据存放在同一个表空间中。

对于物化查询表，（MQT）和其他数据库对象，这类对象通常会被存放在独立的表空间上，这个表空间横跨所有数据 BPU。

10.3.10　数据压缩

数据仓库中面对爆发性增长的海量数据，你会选择如何处理？增加外部存储设备吗？增加存储设备可以解决这个问题，但是需要大量资金投入。可以试想一下有没有其他更有效、更经济的方法。

在很多实际场景中，有时候会发现这些海量数据存在大量冗余。如果将这些冗余的数据用数据字典中的数据代替，就会减少很多空间占用，这就是数据压缩。DB2 支持数据行级别的压缩，它采用基于静态数据字典的压缩原理对数据行进行压缩。在行级别的压缩中，如果一行中多个列值的匹配模式是重复的，则 DB2 会使用较短的标记字符串进行替换。

在数据仓库中，强烈推荐使用数据压缩功能，这可以带来很多好处。

首先，可以降低系统对存储空间的要求。通常，表的压缩比为 50%到 60%，当然这依赖于表中的具体数据。

其次，通过压缩可以提高 I/O 性能。虽然压缩会耗费更多的 CPU 时间来进行压缩或解压缩，但却降低了 I/O 需求。所以，通常对于 CPU 比较空闲而 I/O 存在瓶颈的系统，使用压缩来提高效率。

最后，通过压缩可以节省存储空间，进而节省能源，减少热量的排放，符合绿色环保、持续发展的社会主题。

从 V9.7 版本开始，DB2 开始引入了索引压缩，可以通过表函数 ADMIN_GET_INDEX_COMPRESS_INFO 评估索引的压缩效果。只有在对索引有显著的压缩率时才建议使用。通常，由于唯一性索引的压缩效果不太好，所以不主张对它进行压缩。

此外，在对 DB2 数据库进行备份（BACKUP）时，也可以通过压缩加快备份速度。

10.4　数据仓库实战

有许多用户对搭建数据仓库感觉信心不足，实际上是因为没把握住其中的要点。其实只要掌握在上一节中讲解的设计原则与方法，DB2 数据仓库的搭建就会游刃有余了。

本节将以客户实战环境为基础，结合前面讲的原则与方法，详细介绍搭建数据仓库的具体步骤。

10.4.1　硬件环境

本例中的环境，是通信行业中某公司的真实环境，采用了 3 台 IBM Power 740 服务器，9 台 IBM DS3500 作为外部存储器。具体的硬件配置情况，以及网络拓扑结构如图 10-14 所示。

IBM Power 740 服务器的配置为 128G 内存，两颗 8 核 POWER7 CPU。

IBM DS3500 存储：24 块 300GB 容量的 SAS 标准硬盘，采用 RAID6 磁盘阵列技术。主机与存储之间使用 SAN 网络连接。有关 RAID 技术和 SAN 网络技术不属于本书涉及的范围，有兴趣的读者可以查阅相关文档。

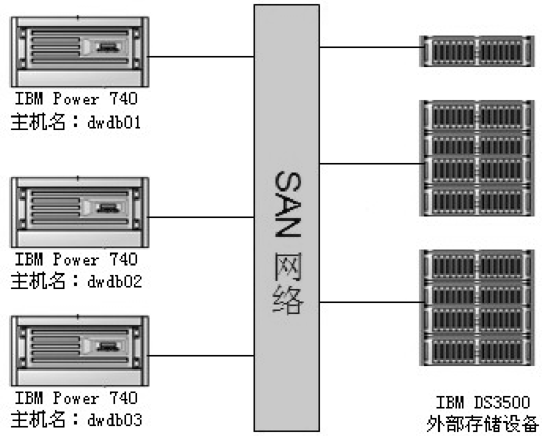


图 10-14　硬件配置和网络拓扑图

根据 10.3 节中的设计原则，结合硬件情况，作出如下的总体规划：

首先，根据对客户数据量和数据仓库工作负载的分析，将一台 Power 740 部署为管理 BCU，设定为分区 0（分区号为 0），承担编目节点和协调节点的功能，并且存储所有非分区数据，从而保证能为客户提供高效的数据交互能力。

其次，在其余的两台 Power 740 上各部署一个数据 BCU，每个 BCU 上规划 8 个数据库分区（也就是数据 BPU），这样，两台 Power 740 上总共有 16 个数据分区，分区号依次为 1~16。

最后，存储采用 RAID6 磁盘阵列技术，这样能为客户提供高效的数据可靠性保证。管理 BCU 使用的三块存储空间，分别用于存储数据、存放日志文件和实例目录；在数据 BCU 使用的每个存储器上划分出两个存储空间，分别为两个数据库分区提供存储服务。更细节的内容请参看下面的具体实现。

10.4.2　实例的规划与实施

完成了对 BCU/BPU 的规划后，下面要完成对数据仓库中实例（Instance）的规划与实施，这一步的工作包括安装 DB2，设置共享实例目录，创建数据库用户和用户组，以及创建实例。具体步骤如下：

（1）安装 DB2

本例中共有三台服务器，分别是 dwdb01、dwdb02 和 dwdb03，以 root 权限分别登录上述三台服务器，运行 db2_install，把 DB2 安装到默认目录/opt/ibm/db2 下。

（2）创建 DB2 主文件系统

以 root 权限登录服务器 dwdb01，创建实例共享目录/db2home，用于存放数据库实例。

注意：为什么需要 db2home 文件系统？

在 dwdb01 上创建的 DB2 数据库用户的主目录放置在文件系统/db2home 中作为共享路径，它会被服务器 dwdb02 和 dwdb03 挂载。这样，这三台服务器可以使用同一个数据库实例。

（3）确认 NFS 是否运行

在 DB2 数据仓库环境中，所有的服务器使用同一个 DB2 主文件系统，所以首先使用 NFS（Net FileSystem）在 dwdb01 上导出（export）路径/db2home，然后在 dwdb02 和 dwdb03 服务

器上挂载（mount）路径/db2home。

先来检查一下 NFS 的状态。在两台机器上分别运行 lssrc –g nfs 命令，确定 NFS 是否在两台服务器上正常运行，命令如下：

```
# lssrc -g nfs
Subsystem        Group           PID          Status
 biod             nfs            217282        active
 rpc.statd        nfs            295060        active
 rpc.lockd        nfs            282780        active
```

可以看到 rpc.statd 和 rpc.lockd 两个进程都在运行。

注意：共享文件系统，不使用 NFS 可以吗？

搭建 DB2 多分区环境（DPF）时，需要在两台服务器上使用共享的文件系统作为数据库的实例目录。NFS 是实现这种文件共享机制的一种方式。

先来看看 NFS 的优势和不足。使用 NFS 实现共享文件系统非常简便，它是操作系统自带的，免费使用。但是 NFS 也具有一定局限性，这是由 NFS 的自身特点决定的。NFS 存在服务端和客户端，当 NFS 的服务端宕机后，DPF 环境就崩溃了；宕机后，重启 NFS 服务时，其他服务器必须卸载（umount）共享的实例目录/db2home，否则会启动失败。

那么还有别的方式吗？实际上除了 NFS 之外，还可以使用 IBM 的并行文件系统 GPFS。GPFS 文件系统的设计目标是使数据文件分布在集群中的所有节点上，并且允许所有节点共享这些数据文件，也允许并行的应用程序从 GPFS 节点组（nodeset）中的任何节点访问这些文件。

GPFS 的各个节点是平等的，具有高性能和可扩展性，同时也具有高可用性和可恢复性，不会因为一个节点宕机影响到整个文件系统，从而比使用 NFS 具有更高的可靠性。

小技巧：不使用共享文件系统，能搭建分区数据库吗？

首先思考一下，共享文件系统的作用是什么？

如果在服务器 dwdb01、dwdb02 和服务器 dwdb03 上创建了一个共享的文件系统/db2home，那么这三台服务器对该共享文件系统均可以访问，也就是说，它们访问的是一模一样的内容。

既然使用共享文件系统就是为了保证 dwdb01、dwdb02 和 dwdb03 这三台服务器访问相同的内容，假如不使用共享文件系统，而是将服务器 dwdb01 上的文件夹原封不动的复制到服务器 dwdb02 和 dwdb03 上，也就达到了同样的效果。

在 DB2 的分区数据库中，数据库用户的主目录/db2home 是共享的，如果把该目录原模原样地复制到其他的服务器上，同样可以启动分区数据库。有一点需要注意的是，如果数据库节点发生改变，需要及时更改并保证所有服务器上目录/db2home 下的文件 db2nodes.cfg 内容保持一致。

最后，不推荐大家使用非共享文件的方式搭建分区数据库。

（4）导出 DB2 主文件系统

在服务器 dwdb01 上导出 DB2 主文件系统“/db2home”。

本例使用的是 AIX 操作系统，输入命令 smit nfs，选择“Network File System（NFS）”并按回车，选择“Add a Directory to Exports List”并按回车，在“Pathname of directory to export”中输入要导出的路径“/db2home”。

在“Hosts allowed root access”中输入服务器 dwdb02 和 dwdb03 的名称或地址。

（5）在 dwdb02 和 dwdb03 服务器上挂载路径/db2home

在服务器 dwdb02 和 dwdb03 上挂载服务器 dwdb01 上导出的文件系统“/db2home”。

输入命令 smit nfs，选择“Network File System （NFS）”并按回车，选择“Add a File System for Mounting”并按回车，在“Pathname of mount point”、“Pathname of remote directory”和“Host where remote directory resides”中分别输入本地的挂载点/db2home，远端的挂载点/db2home，以及远端主机的地址或主机名。

设定“Security method”为“sys”。

设定“/etc/filesystems entry will mount the directory on system restart”为“yes”。

设定“Mode for this NFS file system”为“read-write”。

设定“Mount file system soft or hard”为“soft”。

小技巧：验证各台服务器上/db2home 已经生效

在服务器 dwdb01 的/db2home 中创建文件 abc.txt，然后登录服务器 dwdb02，在/db2home 中也能看到该文件。

在服务器 dwdb02 上删除文件 abc.txt，在主机 dwdb01 的/db2home 中，可以看到 abc.txt 文件已经被删掉。

（6）创建 DB2 使用的用户和用户组

根据规划，系统将只创建一个生产实例 BCUAIX，在该实例下创建数据库 DWDB。每台服务器配置 8 个逻辑分区，两台服务器共有 16 个数据库分区。

本实例中创建的数据库用户和用户组，以及每个用户或用户组的作用，如表 10-2 所示：

表 10-2　用户和用户组的定义

用户名/组名	说明	分类	主目录	ID
db2admgp	实例管理组	Group	-	600
db2fengp	监控组	Group	-	610
db2dasgp	管理服务器组	Group	-	620
Bcuaix	实例用户	User	/db2home/bcuaix	600
db2fenc1	监控用户	User	/db2home/db2fenc1	610
db2das	管理服务器用户	User	/db2home/db2das	620

在三台服务器 dwdb01、dwdb02 和 dwdb03 上，用户名和用户组名完全相同，分别在这三台服务器上运行下面的命令：

```
mkgroup id=600 db2admgp
mkgroup id=610 db2fengp
mkgroup id=620 db2dasgp
```

```
mkuser id=600 pgrp=db2admgp groups=db2admgp home=/db2home/bcuaix bcuaix
mkuser id=610 pgrp=db2fengp groups=db2fengp home=/db2home/db2fenc1 db2fenc1
mkuser id=620 pgrp=db2dasgp groups=db2dasgp home=/db2home/db2das db2das
```

注意：用户 ID 和用户组 ID 也必须相同

除了上面的用户名和用户组名完全相同外，相应的用户 ID 和用户组 ID 也必须相同，这是因为操作系统实际上是以 ID 来识别用户和用户组的。

（7）编辑服务器 dwdb01、dwdb02 和 dwdb03 的/db2home/bcuaix/.rhosts 文件，增加下面的内容，实现三台服务器间在 bcuaix 用户环境下相互信任，这样使得 bcuaix 用户在这三台服务器上互相访问无须再输入密码。

```
dwdb01 bcuaix
dwdb02 bcuaix
dwdb03 bcuaix
```

注意：在所有服务器上必须配置 rsh

rsh 必须在每台机器上都要配置。这样才能保证 dwdb01、dwdb02 或 dwdb03 服务器上的用户可以通过 rsh 访问另外一台服务器。

在服务器 dwdb01、dwdb02 和 dwdb03 上，分别修改文件 /etc/xinetd.d/rsh。如下所示：

```
service shell
{
  socket_type = stream
  wait = no
  user = root
  log_on_success += USERID
  log_on_failure += USERID
  server = /usr/sbin/in.rshd
  disable = no
}
```

（8）创建 DB2 实例

在 dwdb01 上以 root 权限登录，使用下述命令创建实例。DB2 实例创建成功后，在用户 bcuaix 的主目录下会出现 sqllib 的目录。

```
db2icrt -s ese -u db2fenc1  bcuaix
```

（9）分别在服务器 dwdb01、dwdb02 和 dwdb03 上为 DB2 实例指定服务端口。分别编辑文件/etc/services，增加以下内容。

```
DB2c_bcuaix   50000/tcp
DB2_bcuaix    60000/tcp
DB2_bcuaix_END  60016/tcp
```

其中，DB2c_bcuaix 是用做 TCP/IP 通信协议的服务名，端口号是 50000；DB2_bcuaix 和 DB2_bcuaix_END 是为数据库的多分区间提供高速通信的端口定义，DB2_bcuaix 和 DB2_bcuaix_END 分别是起始端口和结束端口，本例中共保留了 17 个端口，为本环境中 17 个数据库分区提供服务。

（10）配置实例 bcuaix 的 db2nodes.cfg，内容如下：

```
0    dwdb01   0
1    dwdb02   0
2    dwdb02   1
3    dwdb02   2
4    dwdb02   3
5    dwdb02   4
6    dwdb02   5
7    dwdb02   6
8    dwdb02   7
9    dwdb03   0
10   dwdb03   1
11   dwdb03   2
12   dwdb03   3
13   dwdb03   4
14   dwdb03   5
15   dwdb03   6
16   dwdb03   7
```

（11）更改实例通讯参数如下：

```
db2 update dbm cfg using SVCENAME DB2c_bcuaix
db2set db2comm = tcpip
```

（12）验证实例是否创建成功。

```
通过下面的步骤，验证一下实例是否创建成功。
① 使用 bcuaix 用户登录服务器 dwdb01，执行命令 db2sampl，创建样本数据库 sample。
② 使用 db2 connect to sample 命令连接到数据库 sample。
③ 使用 select 或其他 SQL 语句操作 sample 数据库中的数据。
④ 使用 db2 drop database sample 删除 sample 数据库。
```

10.4.3　数据库的规划与实施

数据仓库中完成对实例的创建，仅仅完成了第一步，还需要按照 10.3 节中的设计原则，继续对数据仓库中的数据库进行规划和实施，下面让我们开始吧。

第一步，规划管理节点和数据节点的文件系统，以及数据库文件的存储策略。

（1）管理节点的文件系统规划

在整体规划中，划定了管理节点连接一台外部存储，在存储上划分出三个存储空间，分别用于存放数据、存放日志和保存实例数据，具体示意图如下图 10-15 所示。

图 10-15 中文件系统的划分，能够非常明晰地区分各文件系统的职能（其中 bcuaix 是实例名），例如：

/db2fs/bcuaix/NODE0000，用来存放分区 0 上的永久和临时表空间，同时也用来存放数据库的编目文件。

/db2path/bcuaix/NODE0000，用来存放分区 0 的数据库目录。

/db2mlog/bcuaix/NODE0000，用来存放分区 0 的镜像日志文件。

/db2plog/bcuaix/NODE0000，用来存放分区 0 的主日志文件。这里需要注意的是，这个文

件系统使用的是设备 HDISK7，目的是为了将主日志文件与镜像日志文件（存放在设备 HDISK6 中）分布在不同的设备控制器上，这样可以保证当一个设备控制器出现故障时，DB2 仍然有另外一份日志文件可以使用。

/db2home 是实例的共享目录。

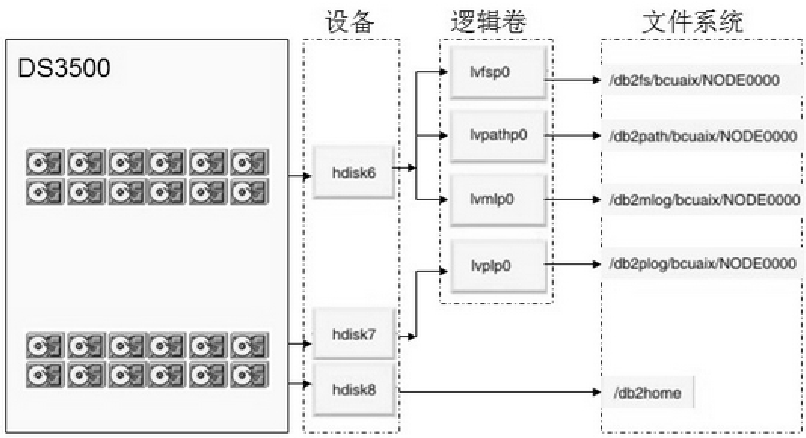


图 10-15　管理节点的文件系统规划

（2）数据节点的文件系统规划

根据对数据节点的规划，每个数据节点连接 4 台 DS3500，每个数据节点上规划了 8 个数据库分区，这样每台外部存储满足两个数据库分区。所有的数据分区都规划有 4 个文件系统（/db2fs/bcuaix/NODENNNN，/db2path/bcuaix/NODENNNN，/db2mlog/bcuaix/NODENNNN，/db2plog/bcuaix/NODENNNN），其中 NNNN 是分区的编号。

与数据分区对应的文件系统的规划，如下图 10-16 所示，每台 DS3500 划分出两个设备。以设备 HDISK7 为例，该设备上的文件系统划分如下：

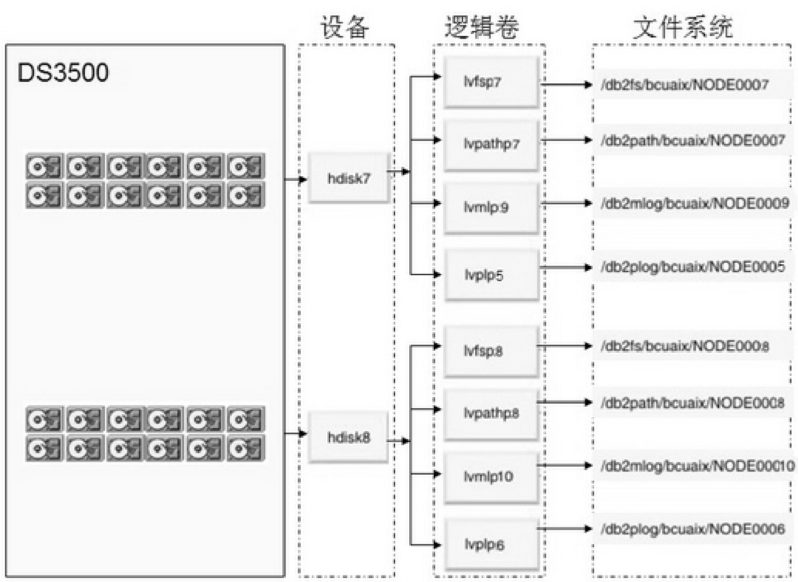


图 10-16　数据节点的文件系统规划

/db2fs/bcuaix/NODE0007，用来存放分区 7 上的永久和临时表空间。

/db2path/bcuaix/NODE0007，用来存放分区 7 的数据库目录。

/db2mlog/bcuaix/NODE0009，用来存放分区 9 的镜像日志文件。

/db2plog/bcuaix/NODE0005，用来存放分区 5 的主日志文件。

需要注意的是，HDISK7 上存放的是分区 7 的数据文件、分区 5 的主日志文件和分区 9 的镜像日志文件，而分区 7 的日志文件放在其他磁盘设备上。这样可以避免读写同一磁盘设备数据文件和日志文件时出现 I/O 竞争。此外，把主日志文件和镜像日志文件存放在不同的磁盘上，是为了确保当某个磁盘出现故障时还有另外一份日志文件可以使用。

第二步，创建数据库、数据库分区组和缓冲池。

（1）创建数据库

下面的命令中，创建了一个名为 dwdb 的数据库，其中/db2fs 指定了存放表空间的位置，/db2path 指定了数据库的创建位置。

```
CREATE DATABASE dwdb ON /db2fs DBPATH ON /db2path USING CODESET GBK TERRITORY CN
```

（2）创建数据库分区组

下面的命令创建了两个数据库分区组，其中 PDPG 用于存放横跨所有数据分区的表，数据分区编号从 1 到 16；SDPG 是单分区的分区组，只包含分区 0。

```
CREATE DATABASE PARTITION GROUP PDPG ON DBPARTITIONNUMS (1 TO 16);
CREATE DATABASE PARTITION GROUP SDPG ON DBPARTITIONNUMS (0);
```

（3）创建和修改缓冲池

下面的命令创建了大小约为 4.5GB 的缓冲池，其中 1.5GB 定义为块区域。

```
CREATE BUFFERPOOL BP_16K ALL DBPARTITIONNUMS SIZE 300000 PAGESIZE 16K
NUMBLOCKPAGES 100000 BLOCKSIZE 16;
```

第三步，在执行完上述操作后，需要重新配置一下数据库日志的存放位置。

（1）使用下面的命令设置默认的 DB2 日志存放位置

```
db2 update db cfg for bcudb using NEWLOGPATH /db2plog/bcuaix
```

（2）镜像日志的存放位置也需要重新配置

```
db2 update db cfg for bcudb using MIRRORLOGPATH /db2mlog/bcuaix;
```

第四步，创建表空间。

（1）创建一个横跨所有数据分区的表空间 TS_BIG

```
CREATE TABLESPACE TS_BIG IN DATABASE PARTITION GROUP PDPG BUFFERPOOL BP_16K
OVERHEAD 4.0 TRANSFERRATE 0.04;
```

（2）创建一个单分区的表空间 TS_SMALL

```
CREATE TABLESPACE TS_SMALL IN DATABASE PARTITION GROUP SDPG BUFFERPOOL BP_16K
OVERHEAD 4.0 TRANSFERRATE 0.04;
```

本例中在创建表空间时，使用了 OVERHEAD 和 TRANSFERRATE 两个参数。通俗地讲，OVERHEAD 值是指磁盘驱动器从接到指令到找到数据所花费的时间（单位是毫秒），TRANSFERRATE 值是指把表空间的一个页从硬盘读入内存花费的时间（单位是毫秒）。优化器进行优化查询时，会结合 OVERHEAD 值和 TRANSFERRATE 值综合考量磁盘 I/O 消耗的时间。使用不同技术参数的硬盘时，对上述两个值的设定也是不同的，用户需要根据实际情况进行设定。如果不指定这两个值，DB2 V9 之后的版本将采用默认值，OVERHEAD 的默认值是 7.5 毫秒，TRANSFERRATE 的默认值是 0.06 毫秒。

10.4.4　参数设置

数据库配置参数对数据仓库的性能有很大的影响。不同的硬件环境、应用特点以及工作负载，决定了数据库配置参数值的差异性。因此，设置 DB2 参数不能照猫画虎，生搬硬套，而应该根据具体情况，设置合适的值，以保证数据仓库的高性能。

以下列出的有关参数配置，只是针对本例环境和应用场景设置的，希望对读者的实际工作提供一些参考。

1. DB2 注册变量的设置

关于注册变量参数的相关知识，请参考本书 7.1.4 节。

（1）指定 TCP/IP 为实例的通信协议。

```
db2set DB2COMM=TCPIP
```

（2）指定 DB2 优化器把包含“NOT EXISTS”查询语句翻译成 anti-join 形式，可以达到提升处理速度的目的。

```
db2set DB2_ANTIJOIN=Y
```

（3）指定 DB2 优化器采用扩展的优化方式提升查询性能。

```
db2set DB2_EXTENDED_OPTIMIZATION=ON
```

（4）指定所有表空间均启用并行 I/O，由于没有指定表空间使用的磁盘数，将使用默认的磁盘数 6。不恰当的硬盘并行数值设置，可能会导致 I/O 性能降低。不同的存储设备和不同的 RAID 划分方法，取得最佳性能的设置值也会不同。

```
db2set DB2_PARALLEL_IO=*
```

对于设置该值的最佳实现方法是，用户在实际环境中使用 Linux/Unix 系统中的磁盘测速工具，测试出 I/O 性能最佳的数值。

2. 实例配置参数的设置

关于实例配置参数的相关知识，请参考本书 7.1.2 节。

（1）设定在实例范围内允许私有排序占用的最大内存容量。

数据仓库系统中，排序、连接等操作需要使用大量的排序堆。通常可以先设定一个初始值，在系统运行过程中通过数据库快照观测内存使用情况再做调整。如下所示：

```
db2 update dbm cfg using sheapthres 1400000
```

（2）让数据库自动管理数据库内部消息通讯所需的缓冲区数目。

```
db2 update dbm cfg using FCM_NUM_BUFFERS AUTOMATIC
```

（3）设定通信端口。

```
db2 update dbm cfg using SVCENAME DB2c_bcuaix
```

（4）指定数据库诊断日志的存放路径。

```
db2 update dbm cfg using DIAGPATH /db2fs/bcuaix/db2dump
```

3. 数据库配置参数的设置

关于数据库配置参数的相关知识，请参考本书 7.1.3 节。

（1）插入或者删除一条记录所生成的锁的大小约为 224 字节，在一个数据分区上向表中插入 10 万条记录时，需要大概 22M 的锁空间（100000×224 字节），需要设定 LOCKLIST 值为 22M/4K = 5600。当前系统 LOCKLIST 设定为 16384，即锁空间大约为 64MB。

```
db2_all "db2 update db cfg for dwdb using LOCKLIST 16384"
```

（2）如果 MAXLOCKS 的值设置的太小，容易导致锁升级，从而影响并发性。本例设置值为 10，可以确保当并发插入 3 万条记录没有提交时，也不会引起锁升级。

```
db2_all “db2 update db cfg for dwdb using MAXLOCKS 10”
```

（3）设置排序堆的大小。

```
db2_all "db2 update db cfg for dwdb using SORTHEAP 35000"
```

（4）数据仓库系统中，大量的插入事务操作会产生大量的日志，如果日志文件的大小设置的比较小，会因归档旧日志、分配新日志文件等操作增大系统开销，进而影响数据库系统的性能。可通过参数 LOGFILSIZ 设置日志文件大小，命令如下：

```
db2_all "db2 update db cfg for dwdb using LOGFILSIZ  12800"
```

（5）主日志是按照 LOGFILSIZ 的大小预分配日志文件的，可以根据硬盘容量和 LOGFILSIZ 的大小设定合理的主文件数量。需要注意的是，如果主文件数量太少，未提交的长事务可能导致日志文件被写满，进而会影响 DB2 的正常运行。副日志文件在本例中不启用。

```
db2_all "db2 update db cfg for dwdb using LOGPRIMARY 50"
db2_all "db2 update db cfg for dwdb using LOGSECOND 0"
```

10.4.5 其他操作

1. 设置操作系统的字符集

应客户要求，数据库使用中文字符集，所以在 bcuaix 用户的.profile 中添加以下内容，用来指定当前服务器在 bcuaix 下使用的字符集，保证 DB2 正常运行。

```
export LC_ALL=zh_CN
```

2. 更新 AIX 的环境设置

为了避免 DB2 的性能受到影响，需要设置 AIX 系统中用户的最大进程数（maxuproc）为 4096。

通过下面命令查看当前 maxuproc 的值，发现当前值为 128。

```
lsattr -El sys0|grep maxuproc
maxuproc        128        Maximum number of PROCESSES allowed per user     True
```

使用下面的命令更改 maxuproc 为 4096，并再次查看。

```
root:/db2home # chdev -l sys0 -a maxuproc='4096'
sys0 changed
root:/db2home # lsattr -El sys0|grep maxuproc
maxuproc        4096       Maximum number of PROCESSES allowed per user     True
```

3. 设置 TCP／IP 网络参数值

为了保证多分区数据库之间高效地通讯，需要在分区数据库所在的所有服务器上更改如下所示的参数。需要注意的是，下列参数所设置的值是推荐的最小值，如果你的系统中某个参数已经设置了一个比下列数值更大的值，可以继续使用该值，无须进行修改。

```
thewall = 65536
sb_max = 1310720
rfc1323 = 1
tcp_sendspace = 221184
tcp_recvspace = 221184
udp_sendspace = 65536
udp_recvspace = 65536
somaxconn = 1024
```

小技巧：如何更改 TCP／IP 网络参数值？

（1）查看当前相关的网络参数，可使用命令 no –a | more。

例如，查看 tcp_sendspace 的当前值：

```
root:/db2home # no -a | grep tcp_sendspace
          tcp_sendspace = 16384
```

（2）设置其中的某项参数，则需要使用 no –o parameter_name = <value>。

例如：设置 tcp_sendspace 的值为 221184，并进行查看：

```
root:/db2home # no -o tcp_sendspace=221184
Setting tcp_sendspace to 221184
Change to tunable tcp_sendspace, will only be effective for future connections

root:/db2home # no -a | grep tcp_sendspace
          tcp_sendspace = 221184
```

4. 开启 Oracle 兼容模式

为了方便用户将已有的 Oracle 应用轻松迁移到 DB2 环境中，在创建数据库前需要进行下述操作，以开启 DB2 的 Oracle 兼容模式。

```
-- 开启兼容 Oracle 的模式
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
-- 重新启动 DB2，创建数据库
db2stop
db2start
db2 CREATE DATABASE DWDB
```

注意：数据仓库中必须开启 Oracle 兼容模式吗？

Oracle 兼容模式可以方便用户将 Oracle 应用迁移到 DB2 环境中。但是，在数据仓库中并非必须开启兼容模式。

如果之前并没有建设过数据仓库，而是需要从头开始搭建一个基于 DB2 的数据仓库，那就没有必要开启 Oracle 兼容模式。

如果之前有 Oracle 数据仓库系统，现在打算将其迁移到 DB2 上，那么可以考虑开启 Oracle 兼容模式。这样，以前基于 Oracle 的数据和应用就可以非常便捷地迁移到 DB2 上了。

10.5　工作负载管理（WLM）

在数据仓库的日常运维工作中，对系统中的重要资源进行监控，对负载过高的相关资源进行调整，就能做到“防患于未然”。

DB2 工作负载管理（WLM）是实现上述目标的有力武器。WLM 用来对工作任务进行实时监控、管理资源和任务优先级，从而避免因工作任务无序竞争而导致的系统性能下降。

这里重点讲解在数据仓库环境下，如何针对复杂查询运用 DB2 WLM 提供的负载管理能力。

10.5.1　管理已知的工作负载

如果一个工作负载的类型可以通过它最初的来源判定，例如由哪个用户运行或者来自哪个应用，这样就可以方便地为工作负载指定优先级别和并行限制。

下面的例子里定义了两个新的服务类，一个具有高优先级，另一个具有低优先级。控制已知工作负载来源的操作步骤如下：

（1）设置默认系统类的优先级别。

如下所示，其中 AGENT PRIORITY 用于定义使用 CPU 的优先级，合理范围为 - 20~20，- 20 为最高优先级，20 为最低优先级，下面设置为 - 20 表示最高优先级：

```
db2inst1@dwdb:~> db2 "ALTER SERVICE CLASS SYSDEFAULTSYSTEMCLASS AGENT PRIORITY -20"
DB20000I  The SQL command completed successfully.
```

注意：AGENT PRIORITY 在 DB2 V10 中不再推荐使用

在 DB2 V10 中，AGENT PRIORITY 不再推荐使用，相应的代替技术是 WLM Dispatcher。这是一种新的调度 CPU 资源的技术，感兴趣的读者可以阅读 DB2 信息中心相关内容。

（2）创建两个服务类，一个用于重要的用户和应用，另一个用于普通的用户和应用。这样一来，就可以将系统中不同的工作负载划分到相应的服务类。

下面的命令中，将来源是“db2bp”、用户是“BCUAIX”的应用定义为高优先级别。将来源是“db2batch”的应用定义为低优先级别。

```
db2inst1@dwdb:~> db2 "CREATE SERVICE CLASS LOW_PRIO";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "CREATE SERVICE CLASS HIGH_PRIO";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "CREATE WORKLOAD HIGH_PRIO APPLNAME('db2bp') SYSTEM_USER
('BCUAIX') SERVICE CLASS HIGH_PRIO";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "CREATE WORKLOAD LOW_PRIO APPLNAME('db2batch') SERVICE CLASS
LOW_PRIO";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "GRANT USAGE ON WORKLOAD HIGH_PRIO TO PUBLIC";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "GRANT USAGE ON WORKLOAD LOW_PRIO TO PUBLIC";
DB20000I  The SQL command completed successfully.
```

注意，用户名需要大写。应用的名字要区分大小，应和命令“LIST APPLICATIONS”中显示出来的“Application name”列一样。

（3）将服务类“HIGH_PRIO”设置为第二高优先级别的类。

注意，用户定义的服务类的优先级别必须低于系统默认的优先级别，在第一步中，系统默认的优先级别为“－20”。所以将自定义的服务类“HIGH_PRIO”的优先级别设置为“－19”。

```
db2inst1@dwdb:~> db2 "ALTER SERVICE CLASS HIGH_PRIO AGENT PRIORITY -19 PREFETCH PRIORITY HIGH"
DB20000I  The SQL command completed successfully.
```

（4）通过下面的命令，将服务类“LOW_PRIO”设置为最低优先级别的类。

```
db2inst1@dwdb:~> db2 "ALTER SERVICE CLASS LOW_PRIO AGENT PRIORITY 20 PREFETCH PRIORITY LOW"
DB20000I  The SQL command completed successfully.
```

（5）设置服务类 LOW_PRIO 的工作负载最大并发查询数为 20。

```
db2inst1@dwdb:~> db2 "CREATE THRESHOLD QUEUE1 FOR SERVICE CLASS \"LOW_PRIO\"
> ACTIVITIES ENFORCEMENT DATABASE
> WHEN CONCURRENTDBCOORDACTIVITIES > 20
> CONTINUE"
DB20000I  The SQL command completed successfully.
```

10.5.2　管理不可预见的工作负载

如果无法从来源区分工作负载的类型，那么可以使用一个估计值来控制这些工作负载。每

当一个工作负载开始在系统中运行时，WLM 会根据完成它所需要的时间和资源估计一个系统开销。基于估计的系统开销，工作负载会被划归于一个服务类中。

具体步骤如下。

（1）设置默认的系统服务类（SYSDEFAULTSYSTEMCLASS）的优先级为最高级别。

```
db2 => alter service class sysdefaultsystemclass agent priority -20 \
db2 (cont.) => bufferpool priority high prefetch priority high
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
```

注意，服务类的优先级别必须低于默认的系统服务类。

（2）创建一个新的服务超类（名为 superclass），并针对大、中、小型的查询任务创建三个子类（名字分别为 highcost_class、middlecost_class 和 lowcost_class）。

```
db2 => create service class superclass
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
db2 => alter workload sysdefaultuserworkload service class superclass
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
db2 => create service class highcost_class under superclass \
db2 (cont.) => collect aggregate activity data extended
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
db2 => create service class middlecost_class under superclass \
db2 (cont.) => collect aggregate activity data extended
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
db2 => create service class lowcost_class under superclass \
db2 (cont.) => collect aggregate activity data extended
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
db2 => alter service class sysdefaultsubclass under superclass collect aggregate
activity data extended
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
```

（3）创建新的工作类，并根据查询时间的长短设定行为集合。

```
db2inst1@dwdb:~> db2 "CREATE WORK CLASS SET CONTROL_COST
(WORK CLASS LowCost WORK TYPE DML FOR TIMERONCOST FROM 0 TO 60000,
WORK CLASS MiddleCost WORK TYPE DML FOR TIMERONCOST FROM 60000 to 2000000,
WORK CLASS HighCost WORK TYPE DML FOR TIMERONCOST FROM 2000000 TO UNBOUNDED)";
DB20000I  The SQL command completed successfully.
```

```
db2inst1@dwdb:~> db2 "CREATE WORK ACTION SET QUERY_COST FOR SERVICE CLASS SUPERCLASS
> USING WORK CLASS SET control_cost
> (WORK ACTION MAP_HighCost ON WORK CLASS HighCost
> MAP ACTIVITY TO HighCost_CLASS,
> WORK ACTION MAP_MiddleCost ON WORK CLASS MiddleCost
> MAP ACTIVITY TO MiddleCost_CLASS,
> WORK ACTION MAP_LowCost ON WORK CLASS LowCost
> MAP ACTIVITY TO LowCost_CLASS)";
DB20000I  The SQL command completed successfully.;
```

（4）为不同资源消耗的工作类创建并发限制。

① 为系统消耗高的类定义并行限制。

```
db2inst1@dwdb:~> db2 "CREATE THRESHOLD limit_concur_long
> FOR SERVICE CLASS HighCost_CLASS under SUPERCLASS ACTIVITIES
> ENFORCEMENT DATABASE
> WHEN CONCURRENTDBCOORDACTIVITIES > 10 AND QUEUEDACTIVITIES UNBOUNDED
> CONTINUE";
DB20000I  The SQL command completed successfully.
```

② 为系统消耗低的类定义并行限制。

```
db2inst1@dwdb:~> db2 "CREATE THRESHOLD limit_concur_middle
> FOR SERVICE CLASS MiddleCost_CLASS under SUPERCLASS ACTIVITIES
> ENFORCEMENT DATABASE
> WHEN CONCURRENTDBCOORDACTIVITIES > 30 AND QUEUEDACTIVITIES UNBOUNDED
> CONTINUE";
DB20000I  The SQL command completed successfully.
```

（5）为开销高的类设置低优先级别。

```
db2inst1@dwdb:~> db2 "ALTER SERVICE CLASS HighCost_CLASS under SUPERCLASS
> PREFETCH PRIORITY LOW AGENT PRIORITY 20";
DB20000I  The SQL command completed successfully.
db2inst1@dwdb:~> db2 "ALTER SERVICE CLASS MiddleCost_CLASS under SUPERCLASS
> AGENT PRIORITY 10";
DB20000I  The SQL command completed successfully.
```

（6）在 DB2 V9.7 中，如果系统开销的估计值不准确，可以创建一个阈值来降低工作负载的优先级别。在下例中，类“LowCost_CLASS”中的查询工作开始时具有高优先级别，如果它消耗超过预估的 CPU 资源，该查询工作将被重新映射到较低的类。

```
db2inst1@dwdb:~> db2 "CREATE THRESHOLD SUPERCLASS_L2M
> FOR SERVICE CLASS LowCost_CLASS under SUPERCLASS ACTIVITIES
> ENFORCEMENT DATABASE PARTITION
> WHEN CPUTIMEINSC > 5 SECONDS CHECKING EVERY 1 SECONDS
> REMAP ACTIVITY TO MiddleCost_CLASS
> LOG EVENT MONITOR RECORD";
DB20000I  The SQL command completed successfully.

db2inst1@dwdb:~> db2 "CREATE THRESHOLD SUPERCLASS_M2H
> FOR SERVICE CLASS MiddleCost_CLASS UNDER SUPERCLASS ACTIVITIES
> ENFORCEMENT DATABASE PARTITION
```

```
> WHEN CPUTIMEINSC > 30 SECONDS CHECKING EVERY 10 SECONDS
> REMAP ACTIVITY TO HighCost_CLASS
> LOG EVENT MONITOR RECORD";
DB20000I  The SQL command completed successfully.;
```

10.5.3 管理并行的 LOAD 工作负载

在使用 LOAD 命令进行数据加载时，对 CPU 资源和 I/O 资源的需求很大，尤其在对 MDC 表和分区表进行数据加载时，需要大量的 CPU 资源和 I/O 资源。

为了避免因为 LOAD 工作抢占数据库过多的资源，进而影响整体系统性能，所以需要限制 LOAD 的并行数量。下面的例子中，将控制 LOAD 操作的并行数量最多设置为 2。接下来讲解一下具体实现。

（1）针对 LOAD 操作，定义一个独立工作类。

```
db2inst1@dwdb:~> db2 "CREATE WORK CLASS SET LOAD_TYPE
> (WORK CLASS LOAD_WC WORK TYPE LOAD)"
DB20000I  The SQL command completed successfully.
```

（2）限定 LOAD 操作的并行数目最多为 2 个，语句如下：

```
db2inst1@dwdb:~> db2 "CREATE WORK ACTION SET CONTROL_LOAD FOR DATABASE
> USING WORK CLASS SET LOAD_TYPE
> (WORK ACTION LIMIT_LOAD ON WORK CLASS LOAD_WC
> WHEN CONCURRENTDBCOORDACTIVITIES > 2 AND QUEUEDACTIVITIES UNBOUNDED
> CONTINUE)"
DB20000I  The SQL command completed successfully.
```

通过上述的设定，可以有效地减少 LOAD 工作对系统资源的占用，从而避免整体系统性能的降低。

10.5.4 监控工作负载

在 DB2 中，可以创建监控器来监控工作负载的运行情况，从而发现潜在的性能问题。监控工作负载的具体步骤如下：

（1）使用下面的语句创建表空间 MONITOR，用于存放 WLM 用到的相关表，这个表空间创建在 IBMDEFAULTGROUP 数据库分区组中。

```
CREATE TABLESPACE MONITOR MAXSIZE 10G;
```

（2）DB2 WLM 中共有三种不同类型的监控方式，分别为：

- 统计事件监控：用于收集所有活动事件的统计信息。
- 活动事件监控：用于收集某个特定类的活动事件的信息，同时也用于收集超越阈值的活动事件的信息。
- 阈值事件监控：用于收集被超越的阈值信息。

使用下面的命令分别创建这三种不同类型的事件监视器用于上述监控。

① 创建统计事件监视器。

```
db2 => create event monitor wlm_stats for statistics \
db2 (cont.) => write to table \
db2 (cont.) => scstats ( table wlm_stats_sc in monitor ), \
db2 (cont.) => wcstats ( table wlm_stats_wc in monitor ), \
db2 (cont.) =>  wlstats ( table wlm_stats_wl in monitor ), \
db2 (cont.) =>  qstats  ( table wlm_stats_q  in monitor ), \
db2 (cont.) =>  histogrambin ( table wlm_stats_histogram in monitor ), \
db2 (cont.) => control ( table wlm_stats_control in monitor ) \
db2 (cont.) => autostart
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
```

② 创建活动事件监视器。

```
db2 => create event monitor wlm_event  for activities write to table \
db2 (cont.) => activity ( table wlm_event in monitor), \
db2 (cont.) => activitystmt ( table wlm_event_stmt in monitor), \
db2 (cont.) => activityvals ( table wlm_event_vals in monitor), \
db2 (cont.) => control ( table wlm_event_control in monitor) \
db2 (cont.) => autostart
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
```

③ 创建阈值事件监视器。

```
db2 => create event monitor wlm_thresholds for threshold violations \
db2 (cont.) => write to table \
db2 (cont.) => thresholdviolations (table wlm_thresh_violations in monitor), \
db2 (cont.) => control ( table wlm_thresh_control in monitor ) \
db2 (cont.) => autostart
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.;
```

（3）创建上述事件监视器后，可以手动启动或停止它们。如果在定义中包含了 AUTOSTART 选项，当相应的数据库分区被重新激活时，该事件监视器会被自动启动。启动和停止事件监视器的命令如下所示。

启动监视器：

```
SET EVENT MONITOR monitor_name STATE 1;
```

停止监视器：

```
SET EVENT MONITOR monitor_name STATE 0;
```

（4）激活 WLM，让它开始工作。在事件监视器被创建后，它们并不能收集信息，需要使用下面的语句启动事件监控，让它们开始收集信息。

① 对于统计事件监控，使用下面的命令启动。在这个例子中，统计信息基于默认的服务类 SYSDEFAULTSUBCLASS。

```
db2 => alter service class sysdefaultsubclass under sysdefaultuserclass \
db2 (cont.) => collect aggregate activity data extended
DB20000I  The SQL command completed successfully.
db2 => commit
DB20000I  The SQL command completed successfully.
```

② 对于活动事件监控，可以运行下述命令，创建如下的阈值，监控所有运行超过 5 分钟的查询。

```
db2 => create threshold monitorevent \
db2 (cont.) => for database activities \
db2 (cont.) => enforcement database \
db2 (cont.) => when activitytotaltime > 5 minutes \
db2 (cont.) => collect activity data with details and values \
db2 (cont.) => continue
DB20000I  The SQL command completed successfully.
```

③ 对于复杂查询，可以预估一个系统开销值作为阈值，这就是阈值监控。命令如下：

```
db2 => create threshold activecost \
db2 (cont.) => for database activities \
db2 (cont.) => enforcement database \
db2 (cont.) => when estimatedsqlcost > 100000 \
db2 (cont.) => collect activity data on all database partitions with details \
db2 (cont.) => continue
DB20000I  The SQL command completed successfully.
```

DB2 WLM 捕获的信息保存在 WLM_EVENT 和 WLM_EVENT_STMT 表中，通过分析这两个表中的数据，DBA 可以发现系统中潜在的问题。

总的来说，DB2 工作负责管理是一个强大并且易于使用的工具，DBA 无须更改应用程序代码，就可以通过它控制复杂的工作负载，从而使多个应用在 DB2 数据库环境中和谐共处。

10.6 数据仓库优化总结

在 DB2 数据仓库的运维中经常会遇到性能问题，这些性能问题与 OLTP 环境相比有什么区别？如何设定优化目标？如何调优？这一节，我们着重讲述上述问题。

10.6.1 与 OLTP 系统的区别

DB2 数据仓库与 OLTP 系统的性能调优不一样，主要表现在以下几个方面：

（1）事务长短。OLTP 环境中的事务通常是短事务，包含多条 SQL 语句，这些语句常常可以在数秒内完成；数据仓库环境中，往往是长事务，只执行少数几条 SQL 语句，但是却需要运行几分钟，甚至几小时。

（2）数据更新时间不同。在 OLTP 环境中，数据更新操作随时都在发生；对于数据仓库系统，数据通常在特定时间窗口（例如夜间）进行更新，往往需要执行大量提取、转换和装载(ETL)操作。

（3）结果集大小不同。OLTP 环境中的查询语句通常只获取少量数据行；数据仓库查询，尤其是那些用来生成报表或进行多维分析的查询可能返回上百万行数据。

（4）查询复杂度不同。OLTP 环境中的查询语句比较简单，通常只访问少数几个表；但是，数据库仓库环境中的查询语句比较复杂，可能包含十几个甚至更多表的关联，包含嵌套的或通用的表达式、递归 SQL、CASE 表达式等。

（5）SQL 语句的生成方式不同。OLTP 环境，通常是自己编写 SQL 语句；在数据仓库环境中，常常由 OLAP 工具生成 SQL 语句，在执行之前很少有机会修改它，DBA 负责设置适当的 DB2 运行环境，让这些查询能够高效运行。

10.6.2　设定优化目标

在进行调优时，不但要追求良好的性能，还要确保方向正确。如果一个查询运行时间很长，但是用户觉得可以接受，那么这个查询也不一定需要优化；相反，一个频繁执行的查询，由于用户期望它能在 1 秒内完成，那么即使它的运行时间只有 5 秒，也会引起用户的抱怨。

对于判断什么地方需要调优，最好听听用户的声音，优先解决用户最关注的查询语句；如果用户没什么抱怨，那么可以花时间调整那些运行时间长和执行频率高的查询语句。

有关如何优化查询语句，请读者阅读本书第 9 章的相关内容。

10.6.3　物理优化

物理优先从 CPU、存储和内存三方面加以调整。

1. 利用多核 CPU 并行性

从 10.2.1 节了解到，DB2 通常运行在 SMP 服务器和 MPP 服务器集群环境，针对这两种环境，要发挥多核 CPU 的并行能力来提升查询性能。

在 SMP 服务器环境上，一颗 CPU 通常都有多个内核，DB2 可以把处理查询所需的工作分割为片段，并行地执行这些片段，这会显著减少查询时间。通过把数据库管理程序参数 intra_parallel 的值设置为 YES，就可以启用 SMP 服务器的查询并行性。

除了能够在 SMP 服务器中并行查询之外，DB2 还可以通过 MPP 服务器集群来 提升查询性能，读者可以按照 10.3 节有关内容做好 MPP 环境下的配置工作。

2. 做好存储规划

关于存储的设计，原则是保证每一个数据 BPU 都有属于自己的一组物理磁盘，并且保证数据仓库中的所有数据对象都均匀地分布在这些磁盘上。这种设计模式，可以保证每个 BPU 中的所有数据对象，都会平衡地进行 I/O 操作。

为了避免数据和日志读写冲突，数据和日志应该放在不同的磁盘上。此外，把主日志文件和镜像日志文件存放在不同的磁盘上，这样当某个磁盘出现故障时还有另外一份日志文件可以使用。

3. 合理配置缓冲池

对于 I/O 密集型的数据仓库工作负载而言，缓冲池越大，性能通常就越好，这是由于随着

缓冲池大小的增加，磁盘 I/O 活动的数量会减少。

但是对 32 位系统而言，内存大小限制为 4GB，为了能够使用更多的内存，建议优先使用 64 位系统。如果 64 位的服务器有大量内存资源，就应该多分配内存资源供 DB2 数据仓库使用。我曾经见过在 32GB 系统内存的服务器上，只给 DB2 配置了 800MB 的缓冲池，这太小了，配置 16~20GB 比较合适。

10.6.4 逻辑优化

逻辑上，要合理利用 DB2 提供的分区、索引、MDC、MQT 等技术来提升性能，具体的优化建议如下。

（1）对大表进行分区

优先对大表进行进行分区，分区时为表选择合适的分区键，可以是单一列，也可以包含多列。表分区的详细介绍请参考本书 5.3 节。

（2）维护数据库编目统计信息

数据库编目统计信息直接影响 DB2 优化器对访问路径决策的选择，只有为优化器提供全面、准确的统计信息，它才能做出合理的访问路径决策。我们可以通过定期运行 RUNSTATS 命令，收集最新的统计信息。

（3）利用索引

对于 OLTP 环境，索引会使得 Insert、Update 和 Delete 操作变慢，所以在创建索引方面往往采取保守的策略，建议索引的个数为 3~5 个。然而，在数据仓库中绝大多数操作是查询，所以创建更多索引是有意义的，通常建议 5~10 个。但是，也不要为数据仓库中的表创建过多索引，否则会影响 ETL 性能。

（4）利用 MDC 特性

数据仓库环境中，常常会大批地获取行，例如获取某客户在过去 5 年在某区域所有的销售记录。这种情况就可以使用 DB2 提供的多维聚簇 MDC 特性，从而可以大批量地获取具有相同维度值的记录。关于 MDC 的详细介绍，请参考本书 5.4 节。

（5）通过 MQT 提升性能

MQT 是通过查询语句定义的一个结果集，在本书 5.5 节对其有详细介绍，它适合优化复杂报表或者分析应用，它有三个优点：

- DB2 可以自动地重写查询以使用 MQT。
- MQT 中的结果集已经建立了，这样在执行查询时可以从 MQT 中直接获取而不需要再动态地构建这个结果集，这样会显著减少执行时间。
- 可以在 MQT 上创建索引，这会进一步提高查询性能。

10.7 与 Oracle 数据仓库比较

Oracle 在面向数据仓库的解决方案中采用的是 RAC 架构，它的全称是 Real Application

Clusters，即“真正应用集群”。下面从几个角度来看看 RAC 的特点。

从逻辑结构上看，与 DB2 的“Share-Nothing”不同的是，RAC 采用的是“Share-Disk”技术，集群中每个节点上的实例（Instance）都访问同一个数据库。这样的结构在扩展性方面会有很多问题，例如随着节点的增多，节点间的通信、协调的复杂程度就呈几何级数上升，管理难度非常大。

从易用性的角度看，RAC 的补丁安装非常复杂；另外，RAC 的实施比较复杂，涉及网络、存储设备、操作系统、集群软件、数据库产品等，一旦出了问题，定位问题非常困难。

从稳定性的角度看，RAC 的稳定性也有很大的问题。例如在双节点的 RAC 环境中，Oracle 的 DBA 有时候会发现两个节点中一个正常运行、而另一个被锁死的情况；也会出现因 RAC 上线导致数据库系统的性能急剧下降；甚至 RAC 由双机运行变成单机运行，双机完全成为摆设的情况也屡见不鲜。

从扩展性的角度看，RAC 可以增加节点提升集群的处理能力。但是所有的节点（包括新增节点）访问同一个共享磁盘集群，这意味着所有的查询都要访问相同的底层存储设备，从而会受到 I/O 瓶颈的限制。

下面，我们给出 DB2 数据仓库与 Oracle 数据仓库的对比总结，如表 10-3 所示：

表 10-3　DB2 与 Oracle 数据仓库技术对比总结

比较内容	DB2	Oracle
采用的架构名称	DPF	RAC
架构特性	真正的 MPP	数据库层的 SMP 集群+共享磁盘的 MPP 存储层
易用性	安装、维护简单，无须调整应用	安装、配置复杂，维护难度大，需要经常进行调整
稳定性	简单、稳定	复杂、稳定性差
扩展性	极易扩展，不存在瓶颈	扩展性无法突破 I/O 瓶颈
总结	专为数据仓库设计的架构	为 OLTP 系统设计的架构，不适合数据仓库

10.8　小结

在本章开篇对数据仓库的基本概念作了简单阐述，对于没有数据仓库经验的读者可以从掌握这些基本概念入手，为透彻地理解本章后续内容作铺垫。随后本章重点介绍了 DB2 数据库在支持数据仓库方面所具有的特性，并详细讲解了面向数据仓库的 DB2 系统设计原则和经验。通过一个实战案例，让读者掌握如何搭建 DB2 数据仓库。本章最后对数据仓库的性能优化进行了讲解，相信可以帮助读者解决日常运维中遇到的性能问题。

第 11 章 DB2 pureScale 集群数据库

什么是 DB2 pureScale?

同样都是 DB2 for z/OS 的追随者，DB2 pureScale 和 Oracle RAC 之间有什么区别?

DB2 pureScale 在国内有没有真实的使用案例？如果有的话是如何规划和实施的？它的性能和用户体验如何?

与单节点的 DB2 相比，DB2 pureScale 在运维上有什么需要 DBA 特别注意的？DB2 DPF 给用户的感觉也是多台服务器的“集群”，pureScale 与之相比有什么不同?

本章将会给出这些问题的答案。

11.1　深入浅出 DB2 pureScale 架构

DB2 数据库提供了垂直（Scale Up）和水平（Scale Out）两种扩展方式。如图 11-1 所示，垂直扩展方式中，通过在单个服务器内增加 CPU、内存以及存储来提升处理能力；在水平扩展方式中，通过增加服务器节点来提升处理能力，DB2 数据仓库（DPF）和 DB2 pureScale 就是水平扩展方式的两种典型。DB2 数据仓库在第 10 章已经讲述过了，本章重点讲述 DB2 pureScale。

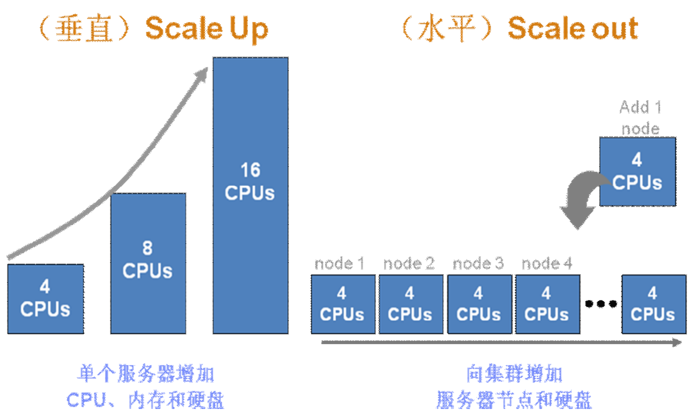


图 11-1　DB2 数据库两种扩展方式

DB2 pureScale 是在 2009 年 12 月发布的 DB2 V9.8 这个版本中首次提出的，它采用了 z/OS 大型机上成熟的共享磁盘（Share Disk）技术，具有非常强的可靠性和稳定性。下面从架构、特点以及和 DB2 DPF 的区别这三个方面分别讲述。

11.1.1　DB2 pureScale 架构

如图 11-2 所示，从架构上看，DB2 pureScacle 分为三层，即数据库集群、集群服务和 GPFS 文件系统。

第一层，数据库集群，由成员和 Coupling Facility 节点（简称为 CF）组成。成员代表一个 DB2 处理引擎，在系统负载变化时可以动态添加或删除成员，数据库集群活跃成员数量的变化不影响客户端的应用。CF 节点采用集中锁机制以保证数据的一致性，另外，CF 节点也用来管理 DB2 数据页的全局缓存。在实际应用中，为了实现高可用性，应配置两个 CF 节点，一主一从，这样可以避免单点故障。

第二层，集群服务。包括 IBM Tivoli Systems Automation for Multiplatforms（TSAMP）和 Reliable Scalable Cluster Technology（RSCT）。TSAMP 技术是用来实现高可用性的。那么 RSCT 技术是用来做什么呢？众所周知，集群中各个节点通过心跳机制来通报彼此的情况，例如，集群中只有 2 个节点，某一时刻节点 1 的心跳出现故障，但是节点 1 还在运行，节点 2 也在运行，但是接收不到节点 1 的心跳了，于是集群就分裂为两个小的集群了，这种场景被称为脑裂（Split-Brain）。那么出现了这种情况，哪个节点来接管整个集群吗？通常需要借助仲裁盘

（Tiebreaker disk）或者仲裁 IP，即由仲裁者来决定由哪个节点来接管。上述脑裂处理的细节都通过 RSCT 技术被自动整合到 DB2 pureScale 中了。

第三层，GPFS 文件系统。DB2 pureScale 各个节点通过 GPFS 文件系统访问共享存储。GPFS 是一个并行的磁盘文件系统，它保证在资源组内的所有节点可以并行访问整个文件系统；而且针对此文件系统的服务操作，可以同时安全地在使用此文件系统的多个节点上实现。关于存储，强烈推荐使用支持“SCSI-3 永久保留”(Persistent Reservations)的存储设备，尽管 DB2 pureScale 也可以支持非 SCSI-3 协议的存储，但是发生故障时，故障节点对支持 SCSI-3 协议的存储进行隔离所需要的时间要远小于非 SCSI-3 协议的存储。为了方便用户，DB2 安装程序集成了 GPFS 文件系统的安装和配置。

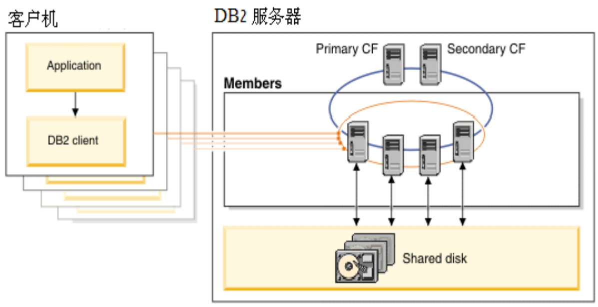


图 11-2　DB2 pureScale 架构

11.1.2　DB2 pureScale 的特点

下面，我们深入讲解一下 DB2 pureScale 的特点。

1. 极好的扩展性

从上节了解到，DB2 pureScale 使用 CF 节点对锁进行集中式管理，从而大大减少了由于不同成员之间进行额外的锁交换请求所带来的开销，这使得随需增加和删除成员成为可能，从而实现了极好的扩展性。如图 11-3 所示，在集群规模达到 112 个成员的情况下，扩展性仍然可以达到 81%以上。

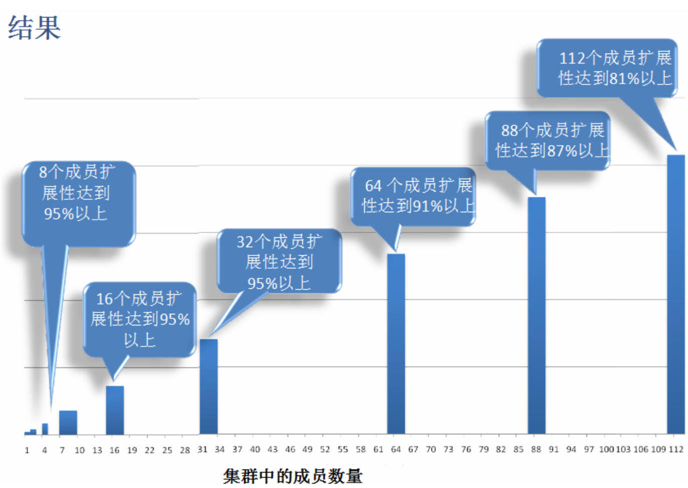


图 11-3　DB2 pureScale 良好的扩展性

2. 应用透明性

对运行在 DB2 pureScale 环境中的应用来说，并不需要知道连接到哪一个成员，根据负载均衡机制，应用的事务会被自动转发给最合适的成员。DB2 pureScale 也实现了原来 DB2 所使用语法的原生支持，这使得这些应用可以不做修改或仅做细小修改就能运行在 DB2 pureScale 环境中。

3. 持续可用性

DB2 pureScale 实现了活跃-活跃式配置，如果一个成员发生宕机时，这时该宕机成员中修改的数据会暂时失效，等其恢复后数据即可用，这个时间很短。在此期间，其余事务会在其他活跃的成员继续进行。

11.1.3 DB2 pureScale 与 DB2 DPF 的区别

在第 10 章讲过 DB2 DPF 技术以及在数据仓库中的应用，那么 DB2 pureScale 和 DB2 DPF 有什么异同呢？

首先，谈谈相似点。二者都是由多台机器组成的数据库集群，DB2 pureScale 由成员和 CF 几点组成，DB2 DPF 由数据节点和协调节点组成；另外，二者都有很好的扩展能力，通过增加节点资源即可线性扩展节点；最后，二者都支持自动客户端路由功能。

下面着重讲述他们之间的不同点。

（1）二者的架构是完全不同的。DB2 DPF 采用非共享（Share Nothing）体系结构，数据库在一个非共享的环境中被分解为独立的分区，每个分区都具有自己的资源，例如内存、CPU、磁盘以及自己的数据索引配置文件和事务日志。节点具有独立处理单一任务的能力，可以独立处理一部分数据，分区间数据通过普通网络即可进行交互。从上节了解到，DB2 pureScale 采用的是共享磁盘架构，成员和 CF 可以直接访问所有磁盘系统，成员和 CF 之间必须通过高速网络进行交互。

（2）二者的用途是完全不同的。DB2 DPF 技术主要用于支持 OLAP 应用，特别是在大型数据仓库方面得到了非常广泛的应用；DB2 pureScale 技术是为 OLTP 应用而量身定做的，侧重于高并发的事务型应用。

（3）二者运行场景下的数据量是不同的。DB2 DPF 由于用在数据仓库方面，所以数据量通常都很大；基于 DB2 pureScale 的应用通常数据量没有 DB2 DPF 应用的场合那么大。

11.2 从细节入手：与 Oracle RAC 对比

通过上面的学习，相信读者已经了解了 DB2 pureScale 的架构以及特点，那么与 Oracle RAC 相比，DB2 pureScale 有什么优势呢？毕竟 Oracle RAC 也是基于共享磁盘技术的，而且取得了极为广泛的应用，作为后来者的 DB2 pureScale 架构上虽与 Oracle RAC 相似，但是具体技术实现细节上却有重大区别。

1. 集群结构

DB2 pureScale 采用 CF 节点提供集中式的锁管理和全局缓存，而 Oracle RAC 采用分布式锁管理和分布式缓存。在 RAC 中，随着集群节点的增多，由分布式架构带来的多节点间的通讯协调复杂度将呈指数级增长，套接字调用、中断和 CPU 上下文切换会大大降低处理效率。在 DB2 puresSale 中，所有成员节点都和 CF 节点单一通讯以获取数据和全局锁。由于采用全局管理锁和缓存，DB2 pureScale 系统结构大大简化了，即使扩展到 128 节点，还能保持比较好的线性扩展能力。

2. 节点故障恢复

当出现成员故障的时候，与 Oracle RAC 相比，DB2 pureScale 的故障恢复时间非常短。

首先来看成员（或实例）故障时，Oracle RAC 集群的恢复时间，如图 11-4 所示，Oracle RAC 集群中实例 1 出现故障，实例 2 将作为恢复实例读取实例 1 的日志并锁定需要恢复的页面，这时所有的 I/O 请求将会被冻结，直到完成对实例 1 的恢复后，集群的 I/O 请求才会被解冻从而整个集群恢复正常。

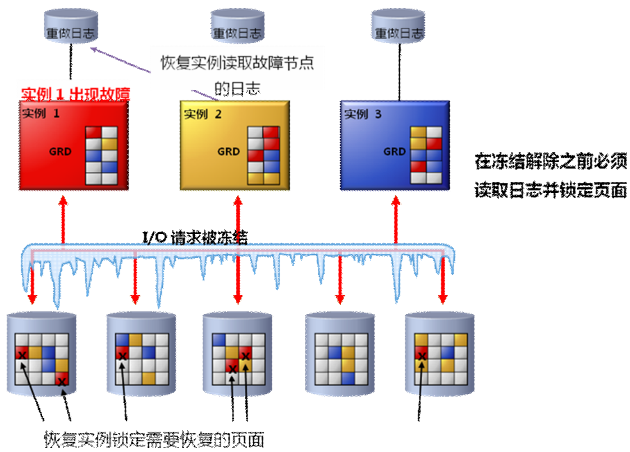


图 11-4　Oracle RAC 成员故障导致 I/O 冻结

如图 11-5 所示，DB2 PureScale 集群中的成员 1 出现故障，成员 2 将负责恢复工作，成员 2 将从 CF 中读取成员 1 需要恢复的页面，不会造成 I/O 冻结，集群很快就能恢复运行。

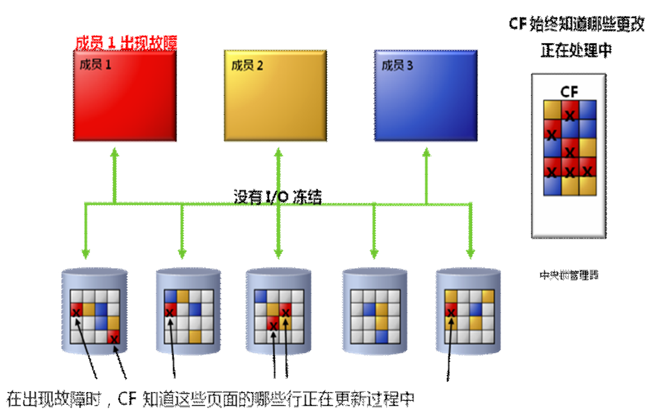


图 11-5　DB2 pureScale 成员故障无 I/O 冻结

3. 节点之间通讯机制

除了节点之间通过心跳测试来检测相互状态外，成员与 CF 之间通过 InfiniBand 网络连接，并使用 RDMA（Remote Direct Memory Access）内存复制直接交换数据，避免高成本的进程间通讯。尽管 InfiniBand 网络也可应用于 Oracle RAC 集群，但是 Oracle RAC 成员之间使用速度较慢的套接字协议进行通信，并且需要一些开销较大的 CPU 中断，从而影响集群性能。

11.3　某银行 DB2 pureScale 实战案例

上面介绍了 DB2 pureScale 的架构，为了让读者更深入理解 DB2 pureScale 在实际生产环境中的规划和实施，接下来分享一个某银行的真实案例。

该银行是中国最大的商业银行，总部位于华东，网点超过 800 家、员工近 6 万人的全国性股份制商业银行，并跻身全球前 100 家大银行之列。

该行的信用卡系统每秒钟要服务的用户量超过了 10000 人，每秒钟会有 5000 笔消费交易，要求 24×7 小时运行，每年允许出现故障的时间是 5 分钟，目前数据量已经达到了 10T 左右，随着数据量和用户量的增加，整个系统的节点要容易扩展。

这对信用卡系统的数据库支撑能力提出了技术挑战，因为原来的基于 IBM AS 400 平台的数据库系统难以满足事务高吞吐量、持续可用性和扩展能力的需要。

最后，经过一系列的技术验证，该银行最终选择了 DB2 pureScale。

11.3.1　从硬件配置开始

经过多次讨论，最终确定了硬件配置方案。主机选用 2 台 IBM Power 740， 存储选择 IBM DS 5300，其原因在于 DS 5300 支持 SCSI-3 持久保护（Persistent Reserve），这样当发生故障时，可以实现快速的 I/O 隔离（I/O Fencing）。单台 Power 740 的配置见表 11-1。

表 11-1　单台 Power 740 主要配置信息

名称	数量	注释
处理器	16 核	
内存	64 G	
HCA 卡（IB 网卡）	2	本实例中，一块用作 IB 网卡，另一块用于连接扩展柜
双端口以太网卡	4	用于连接外部网络
光纤阵列卡	1	用于连接 SAN 存储交换机
内置盘	4	总容量为 300GB

根据项目需要，本案例规划了 2 个成员节点和 2 个 CF 节点，分区规划方案如表 11-2 所示，每台 Power 740 划分成两个 AIX 分区（LPAR），其中一个分区用来运行成员节点，另外一个分区用来运行 CF 节点。

表 11-2　DB2 pureScale 集群服务器资源规划表

物理机器编号	用途	CPU	内存	以太网卡	HCA 卡
P740-01	Member 0	10 核	40 G	2 块	共享一块
P740-01	Primary CF	6 核	24 G	2 块	
P740-02	Member 1	10 核	40 G	2 块	共享一块
P740-02	Secondary CF	6 核	24 G	2 块	

11.3.2　突破难点：网络规划

如图 11-6 所示，DB2 pureScale 集群使用三种网络，分别是以太网络、InfiniBand 网络和 SAN 存储网络。其中以太网络是外部网络，供应用访问数据库集群使用；InfiniBand 网络是内部网络，供集群节点互联使用，之所以选用 InfiniBand 网络是由于其网速非常高，可以达到 40G bit/s，以确保成员和 CF 节点之间高速数据交互；SAN 存储网络供集群节点访问共享存储使用。本案例中，为了避免单点故障，将采用两台 InfiniBand 交换机。

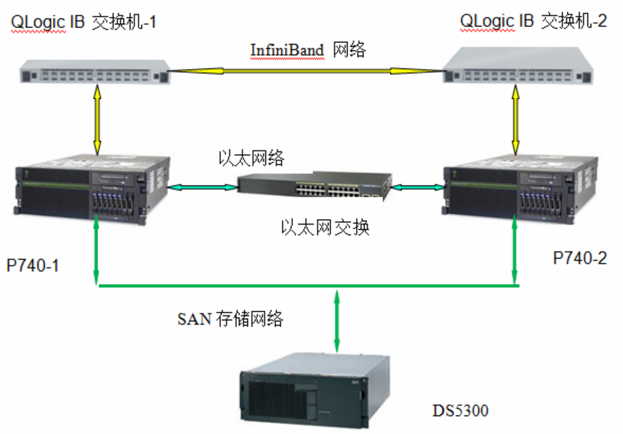


图 11-6　DB2 pureScale 网络拓扑结构

本节将深入探讨以太网规划和 InfiniBand 网络规划，并重点解答配置中常见的问题。

1. 以太网规划和配置

本环境中共有 2 台 Power 740，每台 Power 740 有 4 块网卡，而每台 Power 740 被划分为两个 AIX 分区，每个分区上都分配有 2 块以太网卡，而每个以太网卡是双端口的，所以每个分区一共有 4 个以太网接口可供使用，可以使用 lsdev 命令查看：

```
-bash-3.2# lsdev -Cc adapter  |grep -i 'ent'
ent0    Available 04-00 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent1    Available 04-01 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent2    Available 05-00 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent3    Available 05-01 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
```

为兼顾吞吐量和高可用性，本案例采用 3+1 的方式将每个分区上的 ent0、ent1、ent2 和 ent3 做 EtherChannel 绑定，其中 ent0、ent1 和 ent2 三个网口配置成负载均衡，而使用 ent3 做为它们的备份网口。这样不仅能提升吞吐量，还能够消除以太网卡和以太网交换机出现的单点故障。

在 AIX 操作系统下，可以使用“smitty”交互式界面来完成网卡配置。在命令行提示符下键入“smitty etherchannel”进入交互式界面，选择 ent0、ent1 和 ent2 三个网口绑定在一起，选择 ent3 作为它们的“Backup Adapter”，注意，ent0、ent1 和 ent2 必须与同一台交换机连接，ent3 连接到另外一台交换机上。

在 AIX 系统中做完 EtherChannel 绑定之后，就能看到系统中新增加了一个虚拟网口 ent4，这个 ent4 就是把 ent0、ent1 和 ent2 绑定后系统新加的一个混合端口，如下所示：

```
-bash-3.2# lsdev -Cc adapter  |grep -i 'ent'
ent0   Available 04-00 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent1   Available 04-01 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent2   Available 05-00 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
ent3   Available 05-01 4-Port 10/100/1000 Base-TX PCI-Express Adapter (14106803)
eth4   Available          EtherChannel / IEEE 802.3ad Link Aggregation
```

然后，使用 smitty inet 为 ent4 设置 IP 地址。

重复上面的步骤，在其他所有的分区上完成 Etherchannel 的配置和 IP 地址设定。

最后还需要将 IP 地址和主机名的对应关系加到每个节点的/etc/hosts 文件中去，如下所示：

```
172.16.50.11    db2member0
172.16.50.12    db2member1
172.16.50.15    db2CF0
172.16.50.16    db2CF1
```

2. InfiniBand 网络规划和配置

在这个案例中，使用了两台 InfiniBand 交换机来为 DB2 pureScale 集群提供一个高可用的 InfiniBand 网络环境。

在服务器端，每台 Power 740 上配置有两块双端口的 InfiniBand 网卡，也就是俗称的 IB 卡，其中一块用来连接 Power 740 的扩展柜，因此每台 Power 740 服务器上只能有一块 IB 卡用于 DB2 pureScale 集群。由于每台 740 被划分成两个 AIX 逻辑分区，这两个逻辑分区要共享这块 IB 卡。这块 IB 网卡上有两个端口，很多人可能认为每个分区各占一个端口。实际上不是这样，IB 卡资源的划分是按照带宽来分配的。

在本案例中，成员节点将仅使用这两个端口中的一个；为了提升 CF 节点的吞吐量和可靠性，CF 节点将同时使用这两个端口。

注意：需要为 IB 卡配置 IP 地址

DB2 pureScale 集群中成员节点和 CF 节点之间的通信是通过 RDMA 协议，而不是 IP 网络。但是，仍然需要为 IB 卡配置 IP 地址，该 IP 地址会被 DB2 pureScale 集群管理软件 TSA 用于监控集群内部状态，所以使用私有 IP 地址即可。

在所有 4 个分区上，第一个端口 ib0 使用 10.10.10.*网段的地址，而对 CF 所在分区上的第二个端口 ib1 使用 10.10.11.*网段的地址。最终关于 InfiniBand 网卡的 IP 规划如下表 11-3 所示：

表 11-3　InfiniBand 网络的 IP 规划

机器编号	HCA 卡	用途	端口号	IB 主机名	IB 网络 IP/mask
P740-0	iba0	成员 0	ib0	db2m0-ib0	10.10.10.1/ 255.255.255.0
		CF 0	ib0	db2cf0-ib0	10.10.10.2/ 255.255.255.0
			ib1	db2cf0-ib1	10.10.11.2/ 255.255.255.0
P740-1	iba0	成员 1	ib0	db2m1-ib0	10.10.10.3/ 255.255.255.0
		CF 1	ib0	db2cf1-ib0	10.10.10.4/ 255.255.255.0
			ib1	db2cf1-ib1	10.10.11.4/ 255.255.255.0

根据上面的规划，接下来具体介绍一下在 Power 740 上的安装和配置步骤。

第一步，安装 uDAPL 驱动。

该驱动文件可以从 IBM 官方网站上下载，网址为 https://www14.software.ibm.com/webapp/iwm/web/reg/signup.do?source=aixbp&lang=en_US&S_PKG=udapl。使用 root 用户登录 AIX 系统，使用“smitty install”命令进行安装。安装完成之后用命令“lslpp -l bos.mp64 devices.chrp.IBM.lhca.rte devices.common.IBM.ib.rte udapl.rte”检查驱动是否正确安装，如下所示：

```
    -bash-3.2$ lslpp -l bos.mp64 devices.chrp.IBM.lhca.rte devices.common.IBM.ib.rte
udapl.rte
      Fileset                      Level  State      Description
      ----------------------------------------------------------------------------
    Path: /usr/lib/objrepos
      bos.mp64                      6.1.7.2  COMMITTED  Base Operating System 64-bit
                                                         Multiprocessor Runtime
      devices.chrp.IBM.lhca.rte  6.1.7.1  COMMITTED  Infiniband Logical HCA Runtime
                                                         Environment
      devices.common.IBM.ib.rte  6.1.7.2  COMMITTED  Infiniband Common Runtime
                                                         Environment
      udapl.rte                     6.1.7.1  COMMITTED  uDAPL

    Path: /etc/objrepos
      bos.mp64                      6.1.7.2  COMMITTED  Base Operating System 64-bit
                                                         Multiprocessor Runtime
      devices.chrp.IBM.lhca.rte  6.1.7.1  COMMITTED  Infiniband Logical HCA Runtime
                                                         Environment
      devices.common.IBM.ib.rte  6.1.7.2  COMMITTED  Infiniband Common Runtime
                                                         Environment
      udapl.rte                     6.1.7.1  COMMITTED  uDAPL
```

第二步，在 HMC 中为分区分配 InfiniBand 卡资源。

（1）如图 11-7 所示，打开 HMC 管理界面，选中一个分区的概要，在下拉菜单中选择“配置”→“管理概要文件”

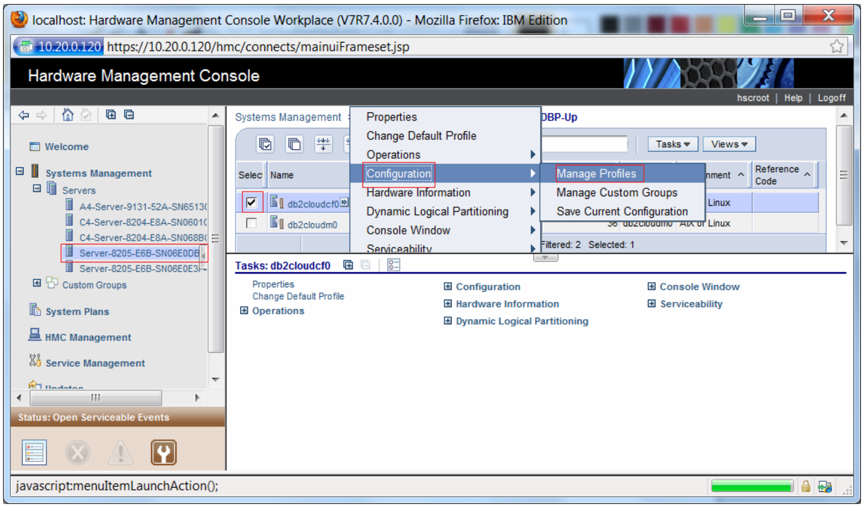


图 11-7　HMC 管理界面

（2）在打开的概要文件中，选择 HCA，可以看到，在本机上有两块 HCA 卡，其中有一块是用来连接扩展柜的，要分配的就是剩下的这一块，即 HMC Managed 属性为“是”的这一块，如图 11-8 所示。

图 11-8　分配 InfiniBand 卡资源

（3）在弹出的 HCA 卡资源分配界面中，可以看到两个可以编辑的选项，一个是 GUID index，这个表示这块 HCA 卡分配给这个分区后的编号，这个编号可以是 1~16 中的任何一个，但是要保证同一物理机器上不同的分区不会出现冲突。在实践中，建议分配的 HCA 卡 GUID 号和分区号保持一致。

另一个是 Capability 选项，该选项表示分配多少资源给该分区，可以有低、中等、高和专用四种分配方式：

- Dedicated(专用)　：分配 100%的资源。
- High(高)：分配 25%的资源。
- Medium(中)：分配 12.5%的资源。
- Low(低)：分配 6.25%的资源。

在本项目中，因为两个分区共享一块 HCA 卡，所以选择 High(高)，如图 11-9 所示。

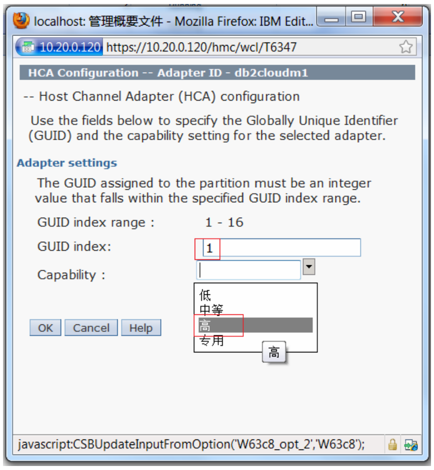


图 11-9 设定属性

（4）编辑完成并保存，随后在该分区上激活该概要文件。重新登录到该分区的操作系统上，用下面的命令可以查看该分区上的 HCA 卡信息：

```
-bash-3.2$ lsdev -Cc adapter  |grep -i ib
iba0    Available      InfiniBand host channel adapter
```

上面的输出表明该分区上的 HCA 卡资源分配成功，重复上面的步骤为剩下的三个分区分配 HCA 卡资源。

第三步，为分区配置 InfiniBand 网络。

首先，需要在每台分区上创建 icm（InfiniBand Communication Manager），在 AIX 系统上使用"smitty icm"创建 icm，如下所示：

```
-bash-3.2# smitty icm
                   Add an Infiniband Communication Manager

Move cursor to desired item and press Enter.

  Add an Infiniband Communication Manager


  +--------------------------------------------------------------------------+
  |               Name IB of Communication Manager to Add                  |
  |                                                                  |
  | Move cursor to desired item and press Enter.                          |
  |                                                                  |
  |   icm                                                             |
  |                                                                  |
  | F1=Help               F2=Refresh             F3=Cancel              |
  | F8=Image              F10=Exit               Enter=Do              |
F1|  /=Find               n=Find Next                                 |
F9+--------------------------------------------------------------------------+
```

使用默认的名称"icm"，设置 icm 相关参数，保持默认即可，可以用命令 lsdev 查看新创

建的 icm：

```
-bash-3.2$ lsdev -C | grep icm
icm       Available           Infiniband Communication Manager
```

创建 icm 完成之后，接下来需要配置 InfiniBand 卡的 IP 地址。大家知道，DB2 pureScale 集群中成员节点和 CF 节点之间的通信是通过 RDMA 协议，而不是 IP 网络。这里配置的 IP 地址被集群管理软件 TSA 用来监控集群状态。在前面也提到过，每一台 Power 740 上的 InfiniBand 卡都有两个端口，而且每个分区都可以同时使用这两个端口。本案例中，将为 CF 节点配置 2 个 InfiniBand 端口，而对成员节点，仅配置 1 个 InfiniBand 端口。

下面讲解一下如何为成员节点配置 IP 地址，步骤如下：

（1） 运行 smitty inet，选择“Change / Show Characteristics of a Network Interface”，在弹出的对话框中选择 ib0。

```
-bash-3.2# smitty inet
                      Network Interface Selection

Move cursor to desired item and press Enter.

  +--------------------------------------------------------------------------+
  |                    Available Network Interfaces                         |
  |                                                                          |
  | Move cursor to desired item and press Enter.                            |
  |                                                                          |
  |  en0   04-00   Standard Ethernet Network Interface                      |
  |  en1   04-01   Standard Ethernet Network Interface                      |
  |  en2   05-00   Standard Ethernet Network Interface                      |
  |  en3   05-01   Standard Ethernet Network Interface                      |
  |  et0   04-00   IEEE 802.3 Ethernet Network Interface                    |
  |  et1   04-01   IEEE 802.3 Ethernet Network Interface                    |
  |  et2   05-00   IEEE 802.3 Ethernet Network Interface                    |
  |  et3   05-01   IEEE 802.3 Ethernet Network Interface                    |
  |  ib0          IP over Infiniband Network Interface                      |
  |  ml0          N/A                                                       |
  |                                                                         |
  | F1=Help              F2=Refresh            F3=Cancel                    |
  | F8=Image             F10=Exit              Enter=Do                     |
F1| /=Find               n=Find Next                                        |
F9
```

（2）配置 ib0 的具体参数，包括“INTERNET ADDRESS (dotted decimal)”、“Network Mask”、“HCA Adapter”和“Adapter’s port number”。IP 地址根据规划表 11-3 进行配置，其余参数使用默认值即可：

```
-bash-3.2# smitty inet
                     Change / Show an IB Interface

Type or select values in entry fields.
Press Enter AFTER making all desired changes.
```

```
[TOP]                                              [Entry Fields]
  Network Interface Name                            ib0
  INTERNET ADDRESS (dotted decimal)                  [10.10.10.1]
  Network MASK (hexadecimal or dotted decimal)         [255.255.255.0]
  IPV6  ADDRESS (colon separated)                   []
  Prefixlength                                   []
  HCA Adapter                                    [iba0]              +
* Adapter's port number                             [1]                 +#
  Partition Key                                  [0xFFFF]             +
  MTU                                          [65532]              +#
  Queue Sizes                                    [4000]               +#
  QKey                                         [0x1E]               +
  Superpacket                                     off                 +
  Interface Specific Network Options
[MORE...8]

F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

成员节点的 IP 地址就配置完了，CF 节点的配置类似，根据规划为其 ib0 端口配置 IP 地址为 10.10.10.2。

为了提升 CF 节点的数据吞吐量，除了配置 ib0 外，还可以同时为其配置另外一个 InfniBand 端口，即 ib1。对于另外一个端口 ib1，需要手动创建，还是用“smitty inet”交互式界面，但选择“Add a Network Interface”，接着选择“Add an IB Network Interface”。在弹出的配置菜单中为新添加的 ib1 端口设置“INTERNET ADDRESS (dotted decimal)”、“Network Mask”、“Network Interface Name”、“HCA Adapter”和“Adapter’s port number”。其中“Network Interface Name”设为 ib1，“HCA Adapter”设为 iba0，“Adapter’s port number”设为 2，表示使用 IB 卡 iba0 的第 2 个端口：

```
-bash-3.2# smitty inet
                         Change / Show an IB Interface

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

[TOP]                                              [Entry Fields]
  Network Interface Name                            ib1
  INTERNET ADDRESS (dotted decimal)                  [10.10.11.2]
  Network MASK (hexadecimal or dotted decimal)         [255.255.255.0]
  IPV6  ADDRESS (colon separated)                   []
  Prefixlength                                   []
  HCA Adapter                                    [iba0]              +
* Adapter's port number                             [2]                 +#
  Partition Key                                  [0xFFFF]             +
  MTU                                          [65532]              +#
  Queue Sizes                                    [4000]               +#
  QKey                                         [0x1E]               +
```

```
    Superpacket                                           off                 +
    Interface Specific Network Options
  [MORE...8]

  F1=Help              F2=Refresh          F3=Cancel           F4=List
  F5=Reset             F6=Command          F7=Edit             F8=Image
  F9=Shell             F10=Exit            Enter=Do
```

这样就完成了第一台 Power 740 的 InfiniBand 网络设置。用类似的方法为第 2 台 Power 740 服务器进行设置。

最后，所有分区的 IP 地址和主机名都要添加到/etc/hosts 文件中，如下所示：

```
172.16.50.11    db2member0
172.16.50.12    db2member1
172.16.50.15    db2CF0
172.16.50.16    db2CF1

10.10.10.1    db2member0-ib0
10.10.10.2    db2CF0-ib0
10.10.11.2    db2CF0-ib1
10.10.10.3    db2member1-ib0
10.10.10.4    db2CF1-ib0
10.10.11.4    db2CF1-ib1
```

第四步，确认 InfiniBand 网络是否成功，检查/etc/dat.conf 文件的配置内容，如下所示：

```
    hca0 u1.2 nonthreadsafe default /usr/lib/libdapl/libdapl.a(shr_64.o) IBM.1.1
"/dev/iba0 1 ib0" " "
    hca1 u1.2 nonthreadsafe default /usr/lib/libdapl/libdapl.a(shr_64.o) IBM.1.1
"/dev/iba0 2 ib1" " "
```

其中，第 1 列 hca0 和 hca1 表示这台机器上使用了 2 个 infiniBand 接口。第 1 行的“/dev/iba0 1 ib0”表示 iba0 上的第 1 个接口 ib0。同理第 2 行的“/dev/iba0 2 ib1”表示 iba0 上的第 2 个接口 ib1。

11.3.3　抓住重点：存储与文件系统

在 DB2 pureScale 集群环境中，GPFS 文件系统将创建在 DS 5300 共享存储上。下面讲述一下配置步骤。

（1）检查多路径驱动软件

为了使 Power 740 服务器识别存储上划分的 vdisk（LUN），需要在服务器上安装多路径驱动。在 Power 平台上，多路径驱动软件为 sddpcm，可以用下面的 lslpp 命令检查是否安装成功：

```
-bash-3.2$ lslpp -l | grep pcm
  devices.sddpcm.61.rte      2.6.1.0  COMMITTED  IBM SDD PCM for AIX V61
```

如果安装成功，可以用命令 pcmpath 查看到设备：

```
-bash-3.2# pcmpath query device
```

```
Total Active/Passive Devices : 39


DEV#:   8 DEVICE NAME: hdisk8  TYPE: 1818     ALGORITHM:  Load Balance
SERIAL: 600A0B80006E668400007E614F582AAA
==========================================================================
Path#     Adapter/Path Name         State     Mode     Select     Errors
    0         fscsi0/path0          OPEN   NORMAL    313871          1
    1         fscsi0/path3          OPEN   NORMAL    310559          0
    2*        fscsi0/path1          OPEN   NORMAL       660          0
    3*        fscsi0/path2          OPEN   NORMAL       660          0

DEV#:   9 DEVICE NAME: hdisk9  TYPE: 1818     ALGORITHM:  Load Balance
SERIAL: 600A0B80006E5E0400007B474F592352
==========================================================================
Path#     Adapter/Path Name         State     Mode     Select     Errors
    0         fscsi0/path0          OPEN   NORMAL    271007          1
    1         fscsi0/path3          OPEN   NORMAL    267261          0
    2*        fscsi0/path1          OPEN   NORMAL       660          0
    3*        fscsi0/path2          OPEN   NORMAL       660          0

DEV#:  10  DEVICE NAME: hdisk10  TYPE: 1818     ALGORITHM:  Load Balance
SERIAL: 600A0B80006E5E0400007B414F582AE0
===========================================================================
Path#     Adapter/Path Name         State     Mode     Select     Errors
    0         fscsi0/path0          OPEN   NORMAL    283951          1
    1         fscsi0/path3          OPEN   NORMAL    279421          0
    2*        fscsi0/path1          OPEN   NORMAL       660          0
    3*        fscsi0/path2          OPEN   NORMAL       660          0

DEV#:  11  DEVICE NAME: hdisk11  TYPE: 1818     ALGORITHM:  Load Balance
SERIAL: 600A0B80006E668400007E724F59235D
===========================================================================
Path#     Adapter/Path Name         State     Mode     Select     Errors
    0         fscsi0/path0          OPEN   NORMAL    288359          1
    1         fscsi0/path3          OPEN   NORMAL    284241          0
    2*        fscsi0/path1          OPEN   NORMAL       660          0
    3*        fscsi0/path2          OPEN   NORMAL       660          0
```

2. 配置 PVID

PVID 是 AIX 系统中用于识别物理卷（PV）的序列号，就类似每个人的身份证号一样。所有用于 DB2 pureScale 的磁盘，包括用于 GPFS 文件系统和仲裁盘的磁盘，在所有的节点上都必须具有相同的 PVID，给磁盘分配 PVID 的步骤如下：

（1）选定一个节点作为当前节点，使用 lspv 命令查看系统上的 hdisk 信息和 PVID 信息。

（2）在当前节点上，给所有的 hdisk 设置 PVID，例如给 hdisk9 设置 PVID，命令如下所示：

```
chdev -l hdisk9 -a pv=yes
```

（3）在其他节点上，通过 rmdev 命令删除所有磁盘原有的 PVID 信息，例如删除 hdisk9 上原有的 PVID 信息，命令如下：

```
rmdev -dl hdisk9
```

（4）在其他节点上运行 cfgmgr 命令，这样系统将从存储中重新检索 PVID，即在第（2）步中设置的 PVID 信息，这样所有磁盘在所有节点上的 PVID 就完全一致了。

3. 文件系统规划

在本次实施过程中，GPFS 文件系统规划如表 11-4 所示，其中 db2fs1 用于实例共享目录，db2data 用于存放数据，db2log 用于存放日志，db2backup 用于存放备份。

表 11-4 GPFS 文件系统规划表

文件系统	挂载路径	用途	容量	逻辑盘
db2fs1	/db2sd_20120419165827	数据库实例共享目录	100G	hdisk2
db2data	/db2fs/db2data	存放数据	3T	hdisk3 hdisk4 hdisk5 hdisk6 hdisk7 hdisk8
db2log	/db2fs/db2log	存放日志	800G	hdisk9 hdiks10
db2backup	/db2fs/db2backup	存放备份	2T	hdisk11 hdisk12 hdisk13 hdisk14
N/A		集群仲裁盘	50M	hdisk15

表 11-4 中最后一行是一个额外的 hdisk15（存储上划分的 LUN，对应主机上的 PV），hdisk15 被用来作为整个 DB2 pureScale 集群的仲裁盘，对空间的需求很小，50M 就够用了。

实际上，DB2 pureScale 集群只需要一个 GPFS 共享文件系统即可安装和运行，但这样的话只能把实例共享目录、数据和日志放在同一个文件系统上。生产环境中，往往考虑把这些目录放在不同的文件系统上以提高性能和便于管理。

11.3.4 检查与配置

本节重点讲述正式安装前的软硬件环境检查和配置。

1. 检查操作系统版本

DB2 pureScale 可以运行在 AIX 6.1 或 AIX 7.1 上，如果是 6.1 建议使用 TL6 SP6。如果使用更早的版本，需要安装不同的操作系统补丁，具体要求请参考 DB2 pureScale 信息中心。

可以用命令“oslevel -s”查看当前安装的操作系统版本，如下所示，其中输出串“6100-06-06-1140”中的“6100”表示 AIX 版本为 6.1，“06-06”表示子版本号为 TL6 SP6。

```
-bash-3.2# oslevel -s
6100-06-06-1140
```

2. 检查 C++运行环境

DB2 pureScale 需要 C++运行环境（xlC runtime environment）的支持，要求的最低版本为 9.0.08。可以用命令 lslpp 查看是否安装了符合条件的 C++运行环境，如下所示，本环境中 C++版本为 11.1.0.2：

```
bash-3.2# lslpp -l xlC.rte
  Fileset                      Level  State      Description
  ----------------------------------------------------------------------------
Path: /usr/lib/objrepos
  xlC.rte                   11.1.0.2  COMMITTED  XL C/C++ Runtime
```

3. 配置 SSH

DB2 pureScale 还需要 OpenSSH 的支持，最低版本要求是 4.5.0.5302，如下所示，本环境中 OpenSSH 版本为 5.8.0.6101：

```
  -bash-3.2# lslpp -la 'openssh*'
  Fileset                      Level  State      Description
  ----------------------------------------------------------------------------
Path: /usr/lib/objrepos
  openssh.base.client     5.8.0.6101  COMMITTED  Open Secure Shell Commands
  openssh.base.server     5.8.0.6101  COMMITTED  Open Secure Shell Server
  openssh.man.en_US      5.8.0.6101  COMMITTED  Open Secure Shell
                                                Documentation - U.S. English

Path: /etc/objrepos
  openssh.base.client     5.8.0.6101  COMMITTED  Open Secure Shell Commands
  openssh.base.server     5.8.0.6101  COMMITTED  Open Secure Shell Server
```

安装了 OpenSSH 之后，接着为 root 用户和实例用户配置 ssh 信任关系，这样各节点相互之间就可以通过 ssh 的方式执行远程命令了。

以 root 用户为例，执行以下步骤配置：

（1）在所有节点上使用 ssh-keygen 命令生成公钥/私钥对：

```
-bash-3.2# ssh-keygen -t dsa
```

该命令执行过程中会提示用户输入信息，直接按回车键，执行完后会在~/.ssh 目录下生成密钥文件 id_dsa（私钥）和 id_dsa.pub（公钥）。

（2）将所有节点上生成的公钥文件内容合并到一个名为 authorized_keys 的文件中，并将 authorized_keys 拷贝到所有节点上的~/.ssh 目录下，并使用命令“chmod 644 authorized_keys”修改其权限。

（3）进行验证。例如，在 db2member0 上执行下面的命令来验证 ssh 信任关系是否配置成功：

```
-bash-3.2# ssh db2member1 date; ssh db2CF0 date;ssh db2CF1 date
Sat Jul  7 16:12:04 BEIST 2012
Sat Jul  7 16:12:03 BEIST 2012
Sat Jul  7 16:12:12 BEIST 2012
```

4. 配置 NTP（Network Time Protocol）

NTP 是用来使计算机之间时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟、GPS 等等)做同步化，它可以提供高精准度的时间校正。

DB2 pureScale 集群中多个成员节点和 CF 节点一起工作，也需要使用 NTP 协议确保这些节点之间的时间同步。下面给出配置 NTP 的具体步骤，其中节点 172.16.50.11 为 NTP 服务器，其他节点为 NTP 客户端，这样客户端和服务器进行同步从而实现所有节点的时间完全一致。

在节点 172.16.50.11 上配置 NTP 服务器的步骤如下：

（1）编辑/etc/ntp.conf 文件，添加如下内容，其中“server 127.127.1.0”表示 NTP 守护进程与其自己的系统时钟同步：

```
-bash-3.2# tail -f /etc/ntp.conf
server 127.127.1.0
driftfile /etc/ntp.drift
tracefile /etc/ntp.trace
```

（2）编辑/etc/rc.tcpip 文件，将以下条目前面的注释符号#去掉，这样 NTP 服务器会自动启动：

```
start /usr/sbin/xntpd -x "$src_running"
```

（3）启动 NTP 服务：

```
startsrc -s xntpd
```

在其他节点上配置 NTP 客户端的步骤如下：

（1）编辑/etc/ntp.conf 文件，指定 NTP 服务器地址为 172.16.50.11，如下所示：

```
-bash-3.2# tail -f /etc/ntp.conf
server 172.16.50.11
driftfile /etc/ntp.drift
tracefile /etc/ntp.trace
```

（2）编辑/etc/rc.tcpip 文件，将以下条目前面的注释符号#去掉，这样 NTP 服务器会自动启动：

```
start /usr/sbin/xntpd -x "$src_running"
```

（3）启动 NTP 服务：

```
startsrc -s xntpd
```

（4）使用 ntpdate 命令和 NTP 服务器完成同步：

```
ntpdate -d 172.16.50.11
```

5. 启用 IOCP（I/O Completion Ports）

在 AIX 平台上启用 IOCP 后，可用来处理 AIX 操作系统上的异步 I/O（AIO）请求，从而改进整体系统性能。启用 IOCP 的步骤如下：

输入 lslpp 命令，检查系统上是否已安装了 IOCP 模块。

```
-bash-3.2# lslpp -l bos.iocp.rte
  Fileset                      Level  State      Description
  ----------------------------------------------------------------------------
Path: /usr/lib/objrepos
  bos.iocp.rte               6.1.7.1  COMMITTED  I/O Completion Ports API

Path: /etc/objrepos
  bos.iocp.rte               6.1.6.0  COMMITTED  I/O Completion Ports API
```

输入 lsdev 命令，检查 IOCP 端口的状态是否为“可用”（Available）。

```
$ lsdev -Cc iocp
```

结果输出应该与以下示例匹配：

```
iocp0 Available  I/O Completion Ports
```

如果 IOCP 端口状态为“已定义”（Defined），那么将状态更改为“可用”（Available），具体步骤如下：

首先，以 root 用户身份登录至服务器并发出以下命令：

```
# smitty iocp
```

随后，选择更改/显示 I/O 完成端口的特征。将已配置的系统重新启动时的状态由“已定义”（Defined）更改为“可用”（Available）。

最后，再次输入 lsdev 命令以确认 IOCP 端口的状态已更改为“可用”（Available）。

小技巧：为所有 DB2 运行环境开启 IOCP

不仅仅是在 DB2 pureScale 环境开启 IOCP 可以提升性能,在单节点单分区环境以及 DPF 环境开启 IOCP 都可以提升 IO 性能，所以建议开启它。

6. 创建组和用户

同普通的 DB2 实例一样，DB2 pureScale 集群实例也需要有实例用户和 Fenced 用户，而且要确保这两个用户在所有的成员和 CF 节点上都存在，并且具有同样的 uid 和 gid。本案例中，使用下面的实例用户和 Fenced 用户，如表 11-5 所示：

表 11-5　创建组和用户

用户类型	用户名	组名
实例用户	db2sdin1	db2iadm1
Fenced 用户	db2sdfe1	db2fadm1

下面是具体步骤。

第一步，创建用户组：

```
mkgroup id=999 db2iadm1
mkgroup id=998 db2fadm1
```

第二步，创建实例用户和 Fenced 用户：

```
mkuser id=1004 pgrp=db2iadm1 groups=db2iadm1 home=/db2home/db2sdin1 db2sdin1
mkuser id=1003 pgrp=db2fadm1 groups=db2fadm1 home=/db2home/db2sdfe1 db2sdfe1
```

第三步，设置实例用户和 Fenced 用户的密码：

```
passwd db2sdin1
passwd db2sdfe1
```

在所有节点上执行上面的命令。

11.3.5　正式安装

1. 安装 DB2 pureScale 软件

所有的准备工作完成之后，就可以开始正式安装 DB2 pureScale 了，本案例选择的 DB2 版本为 10.1，只要选择一个节点作为初始安装节点即可，本案例选择 db2member0 节点。

小技巧：使用 db2prereqcheck 命令进行安装前的检查

DB2 pureScale 的安装过程不复杂，但是对主机、网络和存储有要求，为了防止正式安装时出错，可以使用 db2prereqcheck 命令进行检查，命令格式如下：

bash-3.00# ./db2prereqcheck -p -v 10.1.0.0

在 db2 member0 节点上，使用 ROOT 用户运行安装包中的“db2_install”命令进行安装，根据提示输入一些必要信息，比如安装目录等，具体命令如下：

```
./db2_install
```

整个安装过程大概需要几分钟，TSA 和 GPFS 软件包会在这个过程中一并安装。成功安装后，可以使用 db2ls 命令来查看：

```
-bash-3.2$ /usr/local/bin/db2ls

Install Path                    Level    Fix Pack   Special Install Number   Install
Date                Installer UID
---------------------------------------------------------------------------------
-------------------------------------
/opt/IBM/db2/V10.1              10.1.0.0       0                             Fri Apr 20
11:06:11 2012 BEIST            0
```

2. 创建实例

首先，根据 11.3.3 节的文件系统规划，使用 hdisk2 来创建 DB2 pureScale 共享目录的 GPFS 文件系统：

```
./db2cluster_prepare  -instance_shared_dev /dev/hdisk2
```

该命令执行完成之后会创建一个文件系统，名称自动为 db2fs1，挂载点在/db2sd_

20120419165827，挂载点名称也是自动生成的。接下来可以用 db2icrt 命令来创建 DB2 pureScale 实例：

```
    ./db2icrt -d -a SERVER_ENCRYPT -p 50000 -m db2member0 -mnetdb2member0-ib0 -cf db2CF0
-cfnetdb2CF0-ib0 -instance_shared_dir /db2sd_20120419165827 -tbdev /dev/hdisk15 -s dsf
-u db2sdfe1 db2sdin1
```

该命令各个参数的意义如下：

-d：开启 debug 模式，获得更多的诊断信息。

-a SERVER_ENCRYPT：实例的认证方式为服务器端认证。

-p 50000：实例将采用 50000 作为服务端口。

-m db2member0 -mnetdb2member0-ib0：集群的第一个成员节点，db2member0 为以太网卡接口的主机名，db2member0-ib0 为 InfiniBand 网卡接口的主机名。

-cf db2CF0 -cfnetdb2CF0-ib0：集群的第 1 个 CF 节点，db2CF0 为以太网卡接口的主机名，db2CF0-ib0 为第 1 个 InfiniBand 网卡接口的主机名。

-instance_shared_dir /db2sd_20120419165827：实例的共享目录。

-tbdev /dev/hdisk15：集群的仲裁盘为 hdisk15。

-s dsf：实例的类型为 DB2 pureScale 实例。

-u db2sdfe1：fenced 用户名。

db2icrt 执行完之后，DB2 pureScale 集群实例 db2sdin1 已经被创建出来了，只不过现在集群中只有一个成员节点和一个 CF 节点。接下来需要添加另外的成员节点和 CF 节点。

添加第二个成员节点和 CF 节点：

```
    ./db2iupdt -add -m db2member1 -mnetdb2member1-ib0 db2sdin1
    ./db2iupdt -add -cf db2CF1 -cfnetdb2CF1-ib0 db2sdin1
```

前面提到过，CF 节点可以使用多个 InfiniBand 接口来提高带宽和高可用性，下面为 2 个 CF 节点增加第 2 个 InfiniBand 端口：

```
    ./db2iupdt -update -cf db2CF0 -cfnetdb2CFO-ib0,db2CF0-ib1 db2sdin1
    ./db2iupdt -update -cf db2CF1 -cfnetdb2CF1-ib0,db2CF1-ib1 db2sdin1
```

至此，DB2 pureScale 的集群和实例创建已经完成，可以切换到实例用户 db2sdin1，然后设置 DB2COMM 环境变量：

```
    db2set DB2COMM=tcpip
```

然后运行 db2start 命令启动实例，实例启动成功后可以用“db2instance -list”查看实例状态：

```
    -bash-3.2$ db2instance -list
    ID        TYPE            STATE               HOME_HOST             CURRENT_HOST
ALERT   PARTITION_NUMBER       LOGICAL_PORT    NETNAME
    --        ----            -----               ---------             ------------
-----   ----------------       ------------    -------
    0      MEMBER          STARTED              db2member0              db2member0
NO                0                 0    db2member0-ib0
    1      MEMBER          STARTED              db2member1              db2member1
```

```
NO              0               0    db2member1-ib0
    128    CF           PRIMARY              db2CF0              db2CF0
NO              -               0    db2CF0-ib0
    129    CF             PEER               db2CF1              db2CF1
NO              -               0    db2CF1-ib0

    HOSTNAME              STATE            INSTANCE_STOPPED      ALERT
    --------              -----            ----------------      -----
      db2CF1             ACTIVE                         NO          NO
      db2CF0             ACTIVE                         NO          NO
    db2member1            ACTIVE                         NO          NO
    db2member0            ACTIVE                         NO          NO
```

11.3.6 数据库部署

根据 11.3.3 节的规划，需要创建 3 个 GPFS 文件系统 db2data、db2log 和 db2backup 供 mydb 数据库使用。可以使用 DB2 pureScale 提供的集成命令 db2cluster 来创建这些文件系统，这可以在任何一个成员或 CF 节点上用 root 用户执行。下面是具体的步骤：

（1）创建文件系统 db2data

```
    db2cluster -CFS -CREATE -FILESYSTEM db2data -DISK
/dev/hdisk3,/dev/hdisk4,/dev/hdisk5,/dev/hdisk6 ,/dev/hdisk7 ,/dev/hdisk8 -MOUNT
/db2fs/db2data
```

各个参数的解释如下：

-CFS：表示要使用文件系统管理的相关功能。

-CREATE -FILESYSTEM db2data：新创建 db2data 文件系统。

-DISK /dev/hdisk3,/dev/hdisk4,/dev/hdisk5,/dev/hdisk6,/dev/hdisk7,/dev/hdisk8：文件系统建立在列表中的磁盘上。

-MOUNT /db2fs/db2data：指定文件系统挂载点。

（2）创建文件系统 db2log

```
    db2cluster -CFS -CREATE -FILESYSTEM db2log -DISK /dev/hdisk9,/dev/hdisk10 -MOUNT
/db2fs/db2log
```

（3）创建文件系统db2backup

```
    db2cluster -CFS -CREATE -FILESYSTEM db2backup -DISK
/dev/hdisk11,/dev/hdisk12,/dev/hdisk13,/dev/hdisk14 -MOUNT /db2fs/db2backup
```

（4）创建数据库mydb

```
    db2 create database mydb AUTOMATIC STORAGE YES ON /db2fs/db2data DBPATH ON /db2fs/db2log
USING CODESET UTF-8 TERRITORY CN
```

11.3.7 应用端配置

DB2 pureScale 的工作负载均衡（WLB）机制分为连接级别和事务级别。连接级别的负载均衡发生在客户端建立数据库连接之时，以后不再调整；事务级别的负载均衡发生在事务边界，

需要根据数据库成员服务器的忙闲程度动态进行调整。

在 DB2 pureScale 中，自动客户端重新路由（ACR）功能是指在客户端连接到数据库集群的情况下，如果某个成员发生故障，则会自动地将应用程序重新连接到集群中其他成员上。

如表 11-6 所示，针对 Java 程序和非 Java 程序，DB2 pureScale 对 WLB 和 ACR 默认支持矩阵，例如事务级 WLB 默认情况下都是未激活的。

表 11-6 DB2 pureScale 对 WLB 和 ACR 默认支持矩阵

设置	Java 应用	非 Java 应用
连接级 WLB	不支持	激活
事务级 WLB	未激活	未激活
连接后 ACR	激活	激活
连接前 ACR	未激活	未激活

下面，详细阐述如何针对这些不同应用配置工作负载均衡和自动客户端路由。

1. Java 应用

Websphere 应用服务器（WAS）结合后台 DB2 数据库的应用模式在客户实际生产中广泛使用，那么如果后台数据库是 DB2 pureScale 集群呢？ 在这种环境下，如何来配置呢？

对于 Java 应用程序，事务级别的工作负载均衡缺省是关闭的，连接前自动客户端路由也是关闭的，这样导致客户端在其连接属性中指定的成员服务器宕机时，不会自动路由到其他存活成员服务器。

下面讨论一下如何改变缺省设置，使 WLB 和 ACR 正常工作。

首先，需要在 WAS 服务器的/etc/hosts 文件中添加 pureScale 成员服务器的 IP 地址和主机名：

```
172.16.50.11    db2member0
172.16.50.12    db2member1
```

其次，需要使用 DB2 当前版本自带的 JDBC 驱动程序，之前版本的驱动程序目前不支持工作负载均衡。

最后，如表 11-7 所示，需要在 WAS 应用服务器里进行相关配置。

表 11-7 WLB 和 ACR 在 WAS 中的配置

	DataSource	DriverManager
连接级 WLB	不支持	不支持
事务级 WLB	在“数据源 > 资源名 > 定制属性”增加 enableSysplexWLB 属性：enableSysplexWLB=true	增加 Connection 对象属性：enableSysplexWLB=true

(续表)

	DataSource	DriverManager

连接前 ACR	方法一：增加 DataSource 的属性：
	clientRerouteAlternateServerName=172.16.50.12
	clientRerouteAlternatePortNumber=50000
	方法二：首先生成 JNDI，绑定到 DB2；随后在 WAS 管理控制台中设置 clientRerouteServerListJNDIName 指向所生成的 JNDI。
		方法一：使用 Properties 对象
		Properties props = new Properties();
		props.put("user", "db2sdin1");
		props.put("password", "db2");
		props.put("enableSysplexWLB", "true");
		props.put("clientRerouteAlternateServerName", "172.16.50.12");
		props.put("clientRerouteAlternatePortNumber ", "50000");
		Connection c = DriverManager.getConnection("jdbc:db2://172.16.50.11:50000/mydb", props);
		方法二：在连接串里增加属性
		conn = java.sql.DriverManager
		.getConnection("jdbc:db2://172.16.50.11:50000/HLJRCCB:user=db2sdin1;password=db2;
		enableSysplexWLB=true;clientRerouteAlternatePortNumber=50000;
		clientRerouteAlternateServerName=172.16.50.12;
		");

连接后 ACR	默认为 enable，发生 ACR 时系统会报错误信息，其中 ERRORCODE 为 4498, SQLSTATE 为 08506	默认为 enable，发生 ACR 时系统会报错误信息，其中 ERRORCODE 为 4498, SQLSTATE 为 08506

2. 非 JAVA 应用

多线程 DB2 CLI 程序访问 DB2 pureScale 数据库，如果要实现 WLB 和 ACR 功能，需要配置客户端 db2dsdriver.cfg 文件，该文件存放在客户端的$DB2HOME/sqllib/cfg 目录中。下面是一个 db2dsdriver.cfg 文件的配置例子：

```
<configuration>
 <DSN_Collection>
 <dsn alias="alias1" name="name1"
 host="server1.net1.com" port="50000"/>
 <!-- Long aliases are supported -->
 <dsn alias="longaliasname2" name="name2"
 host="server2.net1.com" port="55551">
 <parameter name="Authentication" value="Client"/>
 </dsn>
 </DSN_Collection>
 <databases>
 <database name="db2member0" host="172.16.50.11" port="50000">
 <parameter name="connectionLevelLoadBalancing" value="true" />
```

```
<WLB>
<parameter name="enableWLB" value="true"/>
<parameter name="maxRefreshInterval" value=30/>
</WLB>
<ACR>
<parameter name="enableAcr" value="true" />
<parameter name="enableSeamlessAcr" value="true" />
<parameter name="enableAlternateServerListFirstConnect" value="true" />
<alternate_server_list>
<server name="db2member1" hostname="172.16.50.12" port="50000" />
</alternate_server_list>
</ACR>
</database>
</databases>
<parameters>
<parameter name="GlobalParam" value="Value"/>
</parameters>
</configuration>
```

前面讲了不同应用是如何配置工作负载均衡和自动客户端路由的，除此以外，生产环境中，例如 SAP 应用、Oracle Tuxedo、IBM CICS 等访问 DB2 pureScale 服务器，可以考虑采用客户端指定（Client Affinity）的方式，这种方式的最大好处是可以为客户端应用指定特定的成员。如图 11-10 所示，每台 CICS 服务器连接到指定的成员服务器，其中 CICS 服务器 1 连接到成员 1，CICS 服务器 2 和 CICS 服务器 3 连接到成员 2。

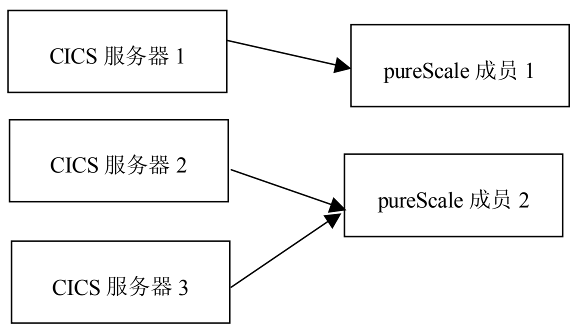


图 11-10　CICS 和 DB2 pureScale 成员服务器连接方式

11.4　DBA 必须掌握的：DB2 pureScale 实用运维命令

曾有人用“警察”的视角来看 DBA 的运维工作。这份警察性质的工作分为两部分：日常维护就是日常巡逻以维护稳定，处理故障就是“反恐”工作以除暴安良。更有趣的是，就像电视剧情节一样，一场温馨甜蜜的婚礼，总要有个恶人蹦出来搞乱场面，而看似平静的日常运维工作总不时会有突如其来的事故，DBA 的工作性质真是透着一个字——“险”。例如当出现紧急事故时，哪怕是在睡梦中，一个电话打来你也得从床上爬起来，用最短的时间解决问题。

实际上，DBA 的运维工作包括日常维护和故障处理，涵盖的具体内容包括软件安装、配置优化、备份策略选择及实施、数据恢复、数据移动、故障排除、性能优化等一系列服务。

上述工作内容也完全适用于 DB2 pureScale 的运维，那么与单节点相比，DB2 pureScale 数据库集群有什么需要特别注意的地方呢？

从某种程度上说，DB2 pureScale 降低了 DBA 的工作量。

首先，单节点 DB2 数据库，因为其自身的 HADR 有很多限制，所以不得不使用操作系统级别的高可用方案，例如 AIX 平台的 HACMP、LINUX 平台的 roseHA，而通常 DBA 不是系统管理员，对操作系统方面的技能未必有那么过硬。DB2 pureScale 就不同了，它本身就具有高可用性，这样就无须 DBA 再去额外配置和管理了。

其次，单节点 DB2 数据库，实现扩展性也是个难题。我见过一些生产系统，刚开始是一个数据库，后来随着用户量和访问量的增加，单节点数据库、单台服务器遇到性能瓶颈了，开始扩展到多个数据库，用多个数据库来承载不同用户或者不同应用的访问，也就是通常说的“应用分库”方案。那么，这些个数据库之间如果要共享数据怎么办？那就只能使用数据库复制技术了。DB2 pureScale 本身提供的扩展性使得上述问题不存在了，随着用户量和访问量的提高，横向扩展成员节点就可以了，非常方便。

最后，单节点 DB2 数据库实现负载均衡更不容易。例如，在上面的“应用分库”方案中，需要应用根据用户名将查询或者事务分发给多个数据库，也就是负载均衡逻辑是由应用来控制的，数据库只是作为数据存储功能而已。这对 DB2 pureScale 来说是很容易实现的，其提供的工作负载均衡能力是非常强大的。

注意：DB2 pureScale 有其自身运维复杂度

> DB2 pureScale 作为集群数据库，除了带来上述运维便捷之处外，也有自身的复杂度。举个简单的例子，一个 DBA 配置和监控单节点单分区的 DB2，明显要比配置和监控 4 个成员、2 个 CF 节点的 DB2 集群要容易得多。那么如果是 16 个成员、2 个 CF 节点呢？这样复杂度就更高了。

为了方便广大读者，我们总结了平时运维中经常用到的一些实用命令（以 mydb 数据库为例），包括管理命令、维护命令和监控命令，希望能起到抛砖引玉的作用。

11.4.1　日常管理命令

1．数据库激活/去激活命令

激活数据库：

```
activate database mydb
```

去激活数据库：

```
deactivate database mydb
```

去激活成员 2 上的数据库：

```
deactivate database mydb member 2
```

2．DB CFG 参数配置命令

在当前成员的机器上执行下述命令，获得当前成员的 DB CFG 信息：

```
get db cfg for mydb
```

修改成员 0 上的 DB CFG 配置参数：

```
update db cfg for mydb member 0 using <parameter name> <parameter value>
```

修改 DB CFG 配置参数，应用到集群所有成员：

```
update db cfg for mydb using <parameter name> <parameter value>
```

3．CF 管理命令

启动 ID 为 129 的 CF 节点：

```
db2start CF 129
```

停止 ID 为 129 的 CF 节点：

```
db2stop CF 129
```

4．成员管理命令

启动编号为 0 的成员节点：

```
db2start member 0
```

停止成员 0：

```
db2stop member 0
```

这时，可以从 sysibmadm.db2_member 管理视图中查看到，成员 0 已停止：

```
select  id,

        varchar(HOME_HOST,10) as home_host,
        varchar(current_host, 10) as cur_host,
        varchar(state, 21) as state,
        alert
from sysibmadm.db2_member
ID          HOME_HOST    CUR_HOST     STATE        ALERT
-----       ----------   -----        -----        -----
    0       NODE101      NODE101      STOPPED      NO
    1       NODE102      NODE102      STARTED      NO
    2 record(s)  selected.
```

5．改变实例中成员顺序

按照下面的命令，首先使用 db2cluster 命令的“-delete-resources”删除资源，随后编辑 db2nodes.cfg，调整成员顺序，最后再用“-create-resources”重建资源。命令如下：

```
db2cluster -cm -delete -resources
DB2 instance resources have been successfully deleted.
//编辑 db2nodes.cfg，重新安排成员顺序
db2cluster -cm -create -resources
All cluster configurations have been completed successfully. db2cluster exiting ...
```

11.4.2　节点维护命令

1．维护某个成员命令

暂停编号为 1 的成员节点：

```
db2stop member 1 quiesce
```

停止 node102 上的 DB2 实例，执行维护，例如升级操作系统补丁：

```
db2stop instance on node102
```

完成维护后，重启 node102 上的 DB2 实例和成员：

```
db2start instance on node102
db2start member 1
```

2．维护某个主机命令

首先停止该主机上访问 GPFS 文件系统的所有进程，随后停止该主机上所有 DB2 实例，最后执行下面的命令：

```
db2cluster -cm -enter -maintenance
db2cluster -cfs -enter -maintenance
```

退出维护状态，在该主机上执行下面的命令：

```
db2cluster -cm -exit -maintenance
db2cluster -cfs -exit -maintenance
```

3．替换 CF 节点命令

停止 ID 为 128 的 CF 节点：

```
db2stop CF 128
```

停止该 CF 节点所在主机的实例：

```
db2stop instance on node101
```

从集群中删除该 CF 节点：

```
db2iupdt -drop -cf node101 db2sdin1
```

向集群中添加新的 CF 节点，ID 为 129：

```
db2iupdt -add -cf  node101:node101-ib1 db2sdin1
```

重启新加入的 CF 节点：

```
db2start CF 129
```

11.4.3 GPFS 文件系统管理命令

1．启动 GPFS 集群

```
db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster -cfs -start -all
All specified hosts have been started successfully.
```

2．停止 GPFS 集群

```
db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster -cfs -stop -all
All specified hosts have been stopped successfully.
```

3．列出当前 GPFS 集群状态

```
db2m0:/ #mmlscluster

GPFS cluster information
========================
  GPFS cluster name:        db2cluster_20120921222531.db2m1
  GPFS cluster id:          12398439165960419547
  GPFS UID domain:          db2cluster_20120921222531.db2m1
  Remote shell command:      /usr/bin/ssh
  Remote file copy command:  /usr/bin/scp

GPFS cluster configuration servers:
-----------------------------------
  Primary server:    db2member1
  Secondary server:  db2CF1

 Node  Daemon node name       IP address     Admin node name       Designation
-------------------------------------------------------------------------------
   1   db2member1              172.16.50.12    db2member1           quorum-manager
   2   db2cf1                  172.16.50.16    db2cf1               quorum-manager
   3   db2cf0                  172.16.50.15    db2cf0               quorum-manager
   4   db2member0              172.16.50.11    db2member0           quorum-manager
```

4．删除 GPFS 集群

```
    db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster -cfs -delete -domain
db2cluster_20120921222531.db2m1
    Deleting the domain db2cluster_20120921222531.db2m1 from the cluster was successful.
```

5. 创建文件系统

例如，下面的命令创建了文件系统 db2test。

```
    db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster  -cfs -create -filesystem
db2test -disk /dev/hdisk30
    File system 'db2test' has been successfully created.
```

6. 列出文件系统所在的磁盘

例如，下面的命令列出了文件系统 db2test 所在的磁盘为/dev/hdisk30。

```
db2m0:/ > db2cluster -cfs -list -filesystem db2test -disk
PATH ON LOCAL HOST                    OTHER KNOWN PATHS
---------------------------------     -------------------------
   /dev/hdisk30
```

7. 删除 GPFS 文件系统

例如，下面的命令删除了文件系统 db2test。

```
db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster -cfs -delete -filesystem db2test
File system 'db2test' has been successfully deleted.
```

8. 挂载 GPFS 文件系统

例如，下面的命令挂载文件系统 db2test，默认挂载路径为/db2fs/db2test。

```
db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster -cfs  -mount  -filesystem
db2test
File system 'db2test' was successfully mounted.
```

9. 卸载 GPFS 文件系统

例如，下面的命令卸载文件系统 db2test。

```
db2m0:/opt/IBM/db2/V10.1FP1_29881/bin #./db2cluster  -unmount -filesystem db2test
File system 'db2test' was successfully unmounted.
```

10. 列出当前集群中文件系统

如下所示，在当前 GPFS 集群中，有 db2data、db2log0、db2log1、db2log2、db2log3 和 db2test 等文件系统。

```
db2m0:/db2test >db2cluster -cfs -list -filesystem
FILE SYSTEM NAME                   MOUNT POINT
---------------------------------     -------------------------
db2backup                         /db2fs/db2backup
db2data                           /db2data
db2fs1                            /db2sd_20120921222623
db2log0                           /db2log/NODE0000/LOGSTREAM0000
db2log1                           /db2log/NODE0000/LOGSTREAM0001
db2log2                           /db2log/NODE0000/LOGSTREAM0002
db2log3                           /db2log/NODE0000/LOGSTREAM0003
db2test                           /db2test
```

11.4.4　监控命令

1．检测主机状态

使用下面的命令获取主机状态，其中 NODE101 为活动（ACTIVE）状态，NODE102 为非活动（INACTIVE）状态：

```
select varchar(hostname, 10) as host, varchar(state,8) as state,varchar(instance_
stopped, 7) as stopped, alert from sysibmadm.db2_cluster_host_state
host     state        stopped      alert
```

```
-----    -----        --------     -----
NODE101  ACTIVE       NO           NO
NODE102  INACTIVE     NO           YES
     2 record(s) selected.
```

2．检测成员状态

使用下面的命令获取成员状态，其中运行在 NODE101 上的成员处于已启动（STARTED）状态，运行在 NODE102 上的成员处于恢复等待（WAITING_FOR_FAILBACK）状态：

```
    select id, varchar(state, 21) as state, varchar(home_host, 10) as home_host,
varchar(current_host, 10) as cur_host, alert from sysibmadm.db2_member
    state                home_host        cur_host     alert
    -----                ---------        --------     ------
    STARTED              NODE101          NODE101      NO
    WAITING_FOR_FAILBACK NODE102          NODE102      YES
        2 record(s) selected.
```

3．检测 CF 状态

使用下面的命令获取 CF 状态，其中 ID 为 128 的 CF 节点处于正在唤醒（CATCHUP）状态，ID 为 129 的 CF 节点显示为主节点（PRIMARY）：

```
    select id, varchar(state, 17) as state, varchar(home_host, 10) as home_host,
varchar(current_host, 10) as cur_host, alert from
TABLE(db2_get_instance_info(NULL,'','','CF',NULL))

    id       state         home_host        cur_host     alert
    ---      ------        ---------        ---------    -----
    128      CATCHUP(85%)  NODE101          NODE101      NO
    129      PRIMARY        NODE102         NOD3102      NO
    2 record(s) selected.
```

4．监控 CF 节点全局缓冲池

使用下面的监控命令，获得 CF 节点的全局缓存信息，例如 ID 为 128 的 CF 节点当前全局缓冲池的大小（current_cf_gbp_size）为 2014 个 4K 页面，已配置的全局缓冲池大小（configured_cf_gbp_size）为 142336 个 4K 页面，目标全局缓冲池（target_cf_gbp_size）为 142336 个 4K 页面：

```
    select substr(host_name, 1, 7) as host, id as hostid, current_cf_gbp_size,
configured_cf_gbp_size, target_cf_gbp_size from table(mon_get_cf(NULL))

    host   hostid current_cf_gbp_size   configured_cf_gbp_size target_cf_gbp_size
    ----    -----    -----------------    ---------------------    -------------------
    node101  128      2014                  142336                   142336
    node102  129      2248                  142336                   142336
        2 record(s) selected.
```

5．监控 CF 节点锁内存

使用下面的监控命令，获得 CF 节点的锁内存信息，例如 ID 为 128 的 CF 节点当前锁内存大小（current_cf_lock_size）为 133852 个 4K 页面，已配置的锁内存大小（configured_cf_lock_size）

为 564224 个 4K 页面，目标锁内存大小（target_cf_lock_size）为 564224 个 4K 页面：

```
    select substr(host_name, 1, 7) as host, id as hostid, current_cf_lock_size,
configured_cf_lock_size, target_cf_lock_size from table(mon_get_cf(NULL))

    host  hostid current_cf_lock_size  configured_cf_lock_size   target_cf_lock_size
    ---- -----    -----------------    ---------------------    -------------------
    NODE101  128  133852               564224                    564224
    NODE102  129  133852               564224                    564224
        2 record(s) selected.
```

6．监控成员节点缓冲池

使用下面的监控命令，从表函数 MON_GET_BUFFERPOOL 中获取结果。其中成员 0 的命中率为 98.19%，成员 1 的命中率为 98.13%：

```
    select bp_name, member, POOL_DATA_GBP_L_READS+POOL_INDEX_GBP_L_READS AS
LOGICAL_READS,
    POOL_DATA_GBP_P_READS+POOL_INDEX_GBP_P_READS AS PHYSICAL_READS,
    ((FLOAT(POOL_DATA_GBP_L_READS+POOL_INDEX_GBP_L_READS)
-FLOAT(POOL_DATA_GBP_P_READS+POOL_INDEX_GBP_P_READS))/FLOAT(POOL_DATA_GBP_L_READS+POO
L_INDEX_GBP_L_READS))
    * 100 AS HIT_RATIO FROM TABLE(MON_GET_BUFFERPOOL('',-2)) WHERE BP_NAME='IBMDEFAULTBP'

    BP_NAME      MEMBER   LOGICAL_READS    PHYSICAL_READS   HIT_RATIO
    ------------ ------ ------------- --------------- ----------
    IBMDEFAULTBP 1        191509           3569             98.13
    IBMDEFAULTBP 0        189314           3419             98.19
        2 record(s) selected.
```

7．监控 CF 节点的 CPU 利用率

使用下面的监控命令，分别获取运行两个 CF 节点所在的 NODE101 和 NODE102 两台主机的 CPU 利用率，其中 NODE101 为 96%，NODE102 为 97%。

```
    select varchar(name, 20) as host_attribute, varchar(value, 25) as values,
varchar(unit,8) as unit from sysibmadm.env_cf_sys_resources

    host_attribute        value                 unit
    HOST_NAME             NODE101                 -
    MEMORY_TOTAL          24108                 MB
    MEMORY_FREE           3504                  MB
    MEMORY_SWAP_TOTAL410  2                     MB
    MEMORY_SWAP_FREE      4063                  MB
    MEMORY_MEM_TOTAL      28211                 MB
    MEMORY_MEM_FREE       7568                  MB
    CPU_USAGE_TOTAL       96                    PERCENT
    HOST_NAME             NODE102                 -
    MEMORY_TOTAL          24108                 MB
    MEMORY_FREE           3342                  MB
    MEMORY_SWAP_TOTAL4102                       MB
    MEMORY_SWAP_FREE      4063                  MB
    MEMORY_MEM_TOTAL      28211                 MB
```

```
MEMORY_MEM_FREE       7406                   MB
CPU_USAGE_TOTAL       97                     PERCENT
     16 record(s) selected.
```

8. 快速查看各个组件状态

DBA 也可以使用 db2instance 命令快速查看集群中各个组件的状态，从下面可以看出，集群中有 2 个成员节点都处于活动状态；另外还有 2 个 CF 节点，一个为主节点，一个为从节点：

```
db2instance -list
ID TYPE STATE HOME_HOST CURRENT_HOST ALERT PARTITION_NUMBER LOGICAL_PORT NETNAME
-- ---- ----- --------- ------------ ----- ---------------- ------------ ---------
0 MEMBER STARTED node101 node101       NO       0                 0        node101
1 MEMBER STARTED node102 node102       NO       0                 0        node102
128 CF   PRIMARY  node101 node101       NO       -                 0        node101
129 CF   PEER     node102 node102       NO       -                 0        node102
HOSTNAME     STATE    INSTANCE_STOPPED ALERT
--------     ------   ---------------- -----
node101      ACTIVE       NO                NO
node102      ACTIVE       NO                NO
```

11.5　DB2 pureScale 规划总结

DB2 pureScale 集群数据库在 DB2 发展中具有里程碑的意义，它满足了关键应用对高可用性、负载均衡和扩展性方面的需求。那么 DBA 在具体的规划和实施中需要注意什么呢？这里，分享一下笔者在为客户做咨询或者实施过程中的经验：

（1）需要几台服务器？至少需要 2 台机器，这样可以防止出现单点故障。一个典型的 Power 平台上的 DB2 pureScale 集群包括 2~4 台服务器，上面创建有多个 AIX 分区（LPAR），每个分区运行 CF 或成员；一个典型的 x86 平台上的 DB2 pureScale 集群包括 4~6 台服务器，每个机器运行 CF 或成员。

（2）硬件选择 Power 平台还是 X86 平台？这个要根据具体应用来定，ERP、银行交易等关键性负载应该选择 Power 平台；对一些只是需要高可用性的普通负载，可以选择 X86 平台。

（3）如何选择 Power 机器型号？在 Power 平台上，可以根据需要，选择 Power 6 中的 550 和 595；也可以选择 Power 7 中的全系列，包括 710、720、730、740、750、770、780 和 795。

（4）如何选择 X86 机器型号？目前 X86 平台可供选择的机器型号只有 3650 M3、3690 X5 和 3850 X5。

（5）如何选择存储型号？只要支持 GPFS 的型号就可以，为了支持失效成员的 I/O 隔离，存储要求最好支持 SCSI-3 持久保护。

（6）如何选择网络？除了 Infiniband 网络外，从 DB2 V10 开始支持万兆以太网。对网络带宽有很高要求的场景下，选择 Infiniband 网络；反之，如果各个成员之间内部互联的带宽要求不高，可以考虑万兆以太网。

（7）如何规划 CF？一个物理机或者一个 AIX 分区（LPAR）上运行一个 CF，需要规划 2

个，一个为主，另一个为从，这样可以避免出现单点故障。运行每个 CF 的 CPU 最大内核数取决于负载，如果是写频繁负载，通常成员内核数和 CF 内核数的比例为 6:1，如果是读频繁负载，这个比例为 10:1。写频繁程度越高，运行每个 CF 需要的内存量就越大。

（8）如何规划成员？一个物理机或者一个 AIX 分区（LPAR）上运行一个成员，后续可以根据需要扩充成员的个数。运行每个成员的 CPU 最大内核数取决于负载的需求，运行每个成员的内存量通常为每个内核 4G~8G 内存。

（9）如何规划存储？存储分为两部分，一部分是一个仲裁盘，容量为 25MB 以上；另一部分是共享盘，用来存放 GPFS 文件系统，其中需要一个实例共享文件系统，容量为 2GB 以上，其余文件系统用来存放数据和日志。

11.6　小结

通过利用先进的硬件架构，DB2 pureScale 将 DB2 for z/OS 上的集中锁和全局缓存技术引入到开放平台。通过对硬件和网络的充分利用，提高了并发性并显著降低了开销，从而提供了更高的扩展能力。

本章首先介绍了 DB2 pureScale 架构，并和 Oracle RAC 进行了对比。随后通过某银行 DB2 pureScale 实战案例，向读者介绍了在 Power 平台上的规划和部署。最后，为了方便读者，提供了 DB2 pureScale 常用运维命令和规划总结。

后记　信念的奇迹

本书的写作，我们是从 2012 年 1 月 1 日正式动笔的，一直持续到 2013 年 5 月 1 日才封笔，总共持续了 480 天。洋洋洒洒 500 多页将近 50 万字的内容，现在终于和广大读者见面了。要谈起中间的写作感受，就如同唐僧师徒四人赴西天取经一般，一路荆棘丛生，历尽千辛万苦、九九八十一难终于取得了真经。我这里不想讲在这本书上夜以继日的写作，牺牲掉了与家人、朋友聚会的时间；我也不想讲这本书经过几十次的版本修订，上百次的会议讨论。我觉得这些都很正常，这是写作中必须付出的投入，并没有值得大讲特讲的。我这里想讲的是这本书的萌发点，实际上是从自己内心深处的某种信念开始的。

2011 年，我几乎满世界飞了近一年，过的是客户、公司和家庭三点一线的生活。做的事非常有挑战性，其重点就是帮助 IBM 公司四处灭火，做了大量数据库迁移、故障诊断和性能优化工作，也给 IBM 的客户和合作伙伴做过大量的培训和技术交流。这样一来在业界和公司内部的名气也逐渐建立起来了。面对成绩，我反复告诫自己，一定要忍得住诱惑，耐得住寂寞，保持一颗平常心。

忙碌之余，有时候静下心来，会反问自己，到底想要什么？其实，这些年来，我内心深处一直有个想法，那就是能为广大数据库工作者编写一本好书。这个想法，从我刚加入 IBM 公司的时候就有了。从那时起，我阅读了大量的数据库经典著作，汲取了大量的技术营养，从内心期望要是有一天自己能写出这样的经典之作该多好啊，但由于工作非常忙碌抽不出写作时间而作罢。时光荏苒，到了 2011 年的时候，我已经在数据库这个领域摸爬滚打了多年，最难打交道的客户都打过交道了，最难相处的合作伙伴也都合作过了，最难优化的 DB2 性能问题也都解决过了，动笔的条件已经基本成熟了。尽管还能为自己找一些理由，比如没有时间，比如技术和经验还需要再积累，但是 IT 行业发展太快，等到什么都准备好的时候，也许掌握的技术和经验已经过时了。此时不动笔，更待何时！

有了动笔的想法还没有付诸实施，但是诱惑来的更快。2011 年 12 月 25 日，在数据库领域一家最有实力的公司多次通过猎头公司诚意邀请我加盟，不仅为我开出了非常有吸引力的薪酬，而且给我提供了很好的位置，可以让我独挡一面，发挥自己所长。这是一个千载难逢的好机会。摆在我面前的是两条路，一条路是选择这家最有实力的数据库公司，获得好的发展机会和优厚待遇，这条路没有悬念，是条宽广之路，前途看好，但是由于公司之间侧重领域不同，编写一本 DB2 方面的经典之作就不可能了；另外一条路是放弃眼前的这个机会，而去实现自己内心深处多年的想法，把自己的优化实战经验和培训经验分享出来，通过书籍的方式帮助广大 DBA 掌握 DB2 设计和性能优化技术，让他们少走弯路，让他们体会到学习数据库技术的快乐。

记得曾经在一本书上看到过这样的话："人生既要追求职业的成功，也要注意内在心灵的成长与提升，才能保持内心的平衡、满足和愉悦"。经过几天的彷徨，我最终义无反顾地放弃了诱惑，坚定地选择了写本好书，于是在 2012 年 1 月 1 日正式动笔了。

能为中国广大 DBA 们写本好书，一直是我的信念。所谓信念，就是心中的一支火把，它能最大限度地燃烧一个人的潜能，指引其飞向梦想的天空。我从不相信运气可以支配个人命运，我从不指望有什么奇迹，只要心中信念的火把不曾熄灭，即使道路再崎岖难行，前途也是一片光明，也一定可以创造信念的奇迹。

王飞鹏

2013 年夏写于上海

缩 略 语

Archive Log（归档日志）

DB2 日志系统的一种工作模式。当最后一个日志文件写满时，会创建一个新的日志文件，这样将来的事务就不会覆盖现有的日志文件。

Backup（备份）

数据库备份就是将数据库中所有数据存储为一个数据副本，供数据库出现问题时恢复使用。

BCU（Balanced Configuration Unit）

BCU 是指把处理器、内存、存储、分区、配置参数等结合在一起，作为一个可扩展的基本单元。它是最小的、可复制的软硬件组合。针对数据仓库系统，BCU 保证了基础架构上的扩展性，提供了磁盘、内存、处理器、网络的优异扩展性能。

Bind（绑定）

DB2 为应用程序的一个或多个 SQL 生成查询访问计划，同时将其和应用程序关联起来的过程。

BPU（Balanced Partition Unit）

BPU 是一个逻辑概念，是指服务于一个 DB2 数据库分区所需的一组相关功能和相关系统资源。

Bufferpool（缓冲池）

缓冲池是内存中的一块存储区域，用于临时读入和更改数据库页（包含表行或索引项）。缓冲池的用途是为了提高数据库系统的性能。

Circular Log (循环日志)

循环日志是 DB2 缺省的日志工作模式。在循环日志中，一旦日志目录中最后一个主日志文件被写满了，就会循环写入，将新的事务写到第一个日志文件中，从而覆盖现有的日志数据。

Container（容器）

容器是存放数据的文件或设备，一个容器只能被单独的表空间使用。

Currently Committed (当前已落实)

从 DB2 V9.7 开始，一种针对游标稳定性（Cursor Stability）隔离级别的特性。当读事务和写事务冲突时，读事务不会被阻塞，而是从日志缓冲区或者活动日志文件中访问冲突行的“当

前已落实”版本，这样就提高了数据库的并发能力。

Cursor（游标）

本质上，游标是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。游标总是与一条 SQL 查询语句相关联，并指向返回的结果集。通过游标，可以从这个结果集中每次提取一条记录。

Data Warehouse（数据仓库）

数据仓库就是面向主题的、集成的、稳定的（不可更新）、随时间变化（不同时间）的数据集合，它可用以支持经营管理中的决策制定过程。

Dead Lock（死锁）

两个或更多事务对锁的循环争用，称为死锁。

DMS（Database Managed Space)

数据库管理表空间，由 DB2 直接管理表空间的使用，其包含的表空间容器可以是文件系统或者裸设备。

DPF（Database Partitioning Feature）

数据库分区功能，一种 DB2 的分区模式。这种分区模式采用 Share-Nothing 非共享体系结构。数据库在一个非共享的环境中被分解为独立的分区，每个分区都具有自己的资源。

EDU（Engine Dispatchable Units）

DB2 中执行特定任务的线程，分为实例级、应用级和数据库级 EDU。

ETL（Extract-Transform-Load）

来源复杂的操作型数据进入数据仓库之前，进行的抽取、转换和加载过程。ETL 的基本流程是，通过消除来自不同数据源数据的不一致之处、以保证数据仓库内的信息一致性。

Event Monitor（事件监视器）

事件监控器收集特定事件发生时的监控数据，例如当一个连接到数据库的应用程序断开时、事务提交时、SQL 语句执行完成时或者死锁发生时等。事件监视器必须由用户创建，而且可以手动地打开和关闭。一旦打开后，事件监视器会不断捕获相关事件的信息。

Explain Tool（解释工具）

查看访问计划和相关信息的 DB2 工具，可用于收集、输出和显示访问计划以及 DB2 生成访问计划时所依据的各种信息。

HADR（High Availability Disaster Recovery）

DB2 高可用性灾难恢复（HADR）提供了一种高可用和灾难恢复解决方案。

Index（索引）

索引是为了快速查找数据而设计的特殊数据结构。索引的优势主要表现在三方面。首先，为表中被请求的数据行提供直接指针；其次，避免了排序操作；第三，避免了对数据表的访问。但索引并不是越多越好，因为在插入、更新和删除时需要额外索引维护开销。

Isolation Level (隔离级别)

在多用户并发环境下，多个事务常常同时执行，但是每个事务都有可能与其他正在运行的事务发生冲突。数据库的隔离级别，是为了保证同时运行的事务具有一致的运行结果。DB2 支持四种隔离级别，分别是可重复读（Repeatable Read）、读稳定性（Read Stability）、游标稳定性（Cursor Stability）和未提交的读（Uncommitted Read）。

Instance (实例)

实例是 DB2 架构中最上层的对象，对于一个数据库安装，我们可以创建多个实例。实例并不直接和数据打交道，它为数据库正常运转提供了各种必要的基础设施，在创建数据库之前，需要先创建实例。

LBAC（Label Based Access Control）

基于标签的访问控制，它允许以细粒度控制谁有权访问单独的行或列。

LOB（Large Object）

大对象，DB2 中一类数据类型，具体包括 BLOB 和 CLOB 两种数据类型。可以用于存储较大的数据对象，最大可以为 2G。

Lock Escalation (锁升级)

在锁资源耗尽后，DB2 会试图把几个行级别的锁合并为一个表级别的锁，从而释放锁空间，这种现象称为锁升级。

Lock Wait（锁等待）

当一个事务在特定数据（例如表或行）上持有锁时，其他事务对该资源的访问都会被拒绝。如果没有某种锁超时检测机制，则事务可能无限期地等待锁的释放，这种情况就叫锁等待。

Lock Timeout（锁超时）

发生锁等待时，为了防止请求锁的应用程序无限期等待，DB2 提供了 LOCKTIMEOUT 数据库参数。这样，当锁等待的时间超过这个参数的设定值，就会触发锁超时事件，并回滚事务。

MDC（MultiDimensional Clustering）

多维群集，一种数据存储方式，它使数据在多个维度上灵活、连续和自动的聚合。通过 MDC，可以提升查询的性能，并且减少了在插入、更新和删除期间对 REORG 和索引维护的需要。

MPP（MassiveParallel Processing）

多机并行处理，狭义上是 DPF 模式的别称。

MQT（Materialized Query Table）

物化查询表，一种将查询结果存储起来的物化表，用于提高复杂查询的效率。每当 MQT 所基于的查询被引用时，数据库将直接从 MQT 的物理存储中得到查询结果。

OLAP (Online Analysis Processing)

联机在线分析处理，数据库负载类型之一，以并发事务较少、查询语句非常复杂、数据更新较少、数据量庞大为特征。

OLTP (Online Transaction Processing)

联机在线事务处理，数据库负载类型之一，以大量事务并发进行、查询语句简单、数据更新操作频繁为特征。

Optimizer（优化器）

优化器是用来产生访问计划的组件，一个 SQL 要以最优性能运行，有赖于优化器产生最优的访问计划。

Package（程序包）

程序包是一个存储过程、静态 SQL 语句或者 UDF 编译后生成的可执行二进制代码。当一个存储过程被编译时，每一条 SQL 编译成一个可执行的节（section），这些节加上其他的一些必要信息组织成一个 DB2 程序包，并存储在 DB2 编目表中。当存储过程被调用时，DB2 从编目表中读取对应的 DB2 程序包直接执行。

Parameter Marker（参数标记）

动态 SQL 语句的一种占位符，作为查询的输入参数。当编译 SQL 语句时可以不指定这些参数的值，而只在执行时临时绑定。使用参数标记避免了对相似 SQL 语句的重复编译。

pureScale

pureScale 是一种高可用性集群数据库解决方案。它利用 z/OS 大型机上经过验证的技术，在开放平台上实现了基于共享磁盘的 DB2 集群。其突出优势表现在几乎无限制的容量、应用透明性和持续可用性。

Query Access Plan

查询访问计划，是 DB2 优化器为 SQL 语句生成的访问数据库数据的方法。

RAID (Redundant Array of Independent Disk)

独立冗余磁盘阵列，一种磁盘组织形式，将多台硬盘组合成单台大容量的硬盘使用。其最大优点是能提高存储系统的传输速率并提供容错功能。

Restore (恢复)

数据库恢复是数据库备份的逆过程，即使用备份介质中的内容覆盖已有数据库。

RCAC（Row and Column Access Control）

行列访问控制，从 DB2 V10 新引入，类似于 LBAC 但使用上更简便。

SMP（Symmetric Multi-Processor）

单机多处理器并行，允许多个 DB2 代理程序并行执行单个查询，从而将复杂 SQL 请求的执行分发到各个处理器的并行方式。

SMS（System Managed Space）

系统管理表空间，依赖于底层操作系统来管理空间分配和使用。

Snapshot Monitor（快照监视器）

快照监视器是 DB2 提供的最常规的监控方式，它可以让你获得包括缓冲池、锁、排序、语句、表等工作对象在一个指定的时间点上的状态映像，这样一旦发现问题就可以立即做初步分析。

System Catalog（DB2 系统编目）

DB2 系统编目由很多表和视图组成，这些表和视图由数据库管理器来维护。在创建一个数据库时，也会创建一组编目表和视图。这些编目描述了数据库对象，例如表、列和索引，并包含关于用户访问这些对象的权限信息。

Table Partitioning（表分区）

表中一个或多个字段作为表分区键列，DB2 根据这些键列的值将表数据分布到多个存储对象（称为数据分区或范围）中。

Table Space（表空间）

表空间是用户定义的存放数据库索引、表数据、临时数据的逻辑空间。一个对象可以存放在多个表空间中，一个表空间含有一个或多个容器。

Transaction（事务）

由一个或多个 SQL 语句组成的工作单元。这些工作单元被视为一个整体，如果都执行成功，执行的结果就在数据库中生效，如果任何一个 SQL 执行失败，则整个工作单元被回滚。

WAL（Write-Ahead Logging）

预写式日志机制，它要求在数据库中更新、插入或删除的数据被写入到磁盘之前，那些记录这些变化的日志必须写入到磁盘。

XQuery

XQuery 是 W3C 所制定的一套标准，用来从 XML 文档中提取信息。针对 XML 的 XQuery 语句非常类似针对关系数据库的 SQL 语句。

参考文献

[1] Hector Garcia-Molina, Jeffery D.Ullman, Jennifer Widom.数据库系统全书.北京：机械工业出版社，2009

[2] JiaWei Han,Micheline Kamber.数据挖掘概念与技术.北京：机械工业出版社，2001

[3] 希尔伯沙茨.数据库系统概念.北京：机械工业出版社，2003

[4] Stephane Faroult,Peter Robson.SQL 语言艺术.温昱，靳向阳.北京：电子工业出版社，2009

[5] Thomas M.Connolly,Carolyn E.Eegg.数据库设计教程.何主洁，黄婷儿.北京：机械工业出版社，2011

[6] Niemiec, Richard J.Oracle Database 10g Performance Tuning Tips & Techniques.McGraw-Hill Osborne Media,2010

[7]《SQL Performance Tuning》Gulutzan, Peter/ Pelzer, Trudy Addison-Wesley,2008

[8] Whei-Jen Chen, Kenneth Chen, Patrick Dantressangle, etc. Oracle to DB2 Conversion Guide:Compatibility Made Easy, IBM Redbooks, 2009

[9] Connor McDonald, Chaim Katz, Christopher Beck, etc. Mastering Oracle Pl/SQL Sprin-ger-Verlag New York Inc, 2005

[10] Serge Rielau SQL PL in DB2 9.7 ALL GROWN UP. IDUG Europe, 2009

[11] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995

[12] IBM DB2 联机文档

朱志辉

IBM资深软件工程师，来自IBM DataStudio开发团队，具有多年的数据库软件开发经验，擅长解决各种数据库应用系统的性能问题，拥有IBM DB2各项认证和Oracle OCP证书。

王富国

IBM Data Warehouse/BI 项目组高级软件工程师，IBM认证高级DBA。有丰富的数据仓库经验，在数据仓库健康检查和高可用性方面也有颇深的研究，具有丰富的性能调优、实施和客户支持经验。

业内热评：

本书完美诠释了DB2性能优化艺术，作者团队全部是IBM中国软件开发中心的资深专家，具有丰富的从Oracle向DB2迁移实施经验，他们的书一定能带领广大的读者实现华丽的从容转身。

——IBM中国开发中心信息管理产品开发部总经理 朱辉

这本书是一帮志同道合之人写就的DB2作品，且不说在IBM公司和数据库社区激起巨大波澜，仅是巨大的销量就足以反映那些或共鸣、或钦佩的心态。作为一名从业20多年的数据库工作者，对书中所阐述的IBM原厂“正规军”的性能优化战略战术深有体会，希望广大读者能从中受益。

——IBM中国软件部Executive IT Specialist 沈刚

这本书讲述了性能优化的方方面面，涵盖物理设计和逻辑设计，涵盖性能监控和参数调整，涵盖SQL语句优化。最难得的是作者分享了两种DB2性能调整方法学，一种是自下而上方法学，强调性能优化要贯穿设计、开发和维护整个数据生命周期，这无疑纠正了很多认识上的误区；另外一种是自下而上方法学，它是作者多年现场摸爬滚打的实战总结。

——IBM中国软件部Consulting IT Specialist 雷林

性能优化往往具有牵涉面广、时间紧的特点，涉及客户、开发商、基础软件、硬件设备等多方面，要求从业人员技术全面且具备良好的沟通协调能力。作者将自己在银行、电信等行业积累的实战案例精彩呈现，让人身临其境，相信能给读者在技术、业务、沟通等各方面带来启发和感悟。

——海通证券股份有限公司信息技术管理部副总经理 王洪涛

本书作者在为银行、电信等行业客户的技术服务中积累了丰富的实战经验，这使得他们能够以广阔的视角来看待性能优化工作。特别是书中有关数据仓库的内容，讲解透彻清晰，字字珠玑，为这本难得的上乘之作锦上添花。

——中信银行资深数据架构师 刘文涛

这本书用实战案例贯穿DB2经典设计和优化技术，不落俗套，角度独特，还特别讲述了DB2 HADR、pureScale等高级架构设计与优化技术，这对于广大读者来讲，大可畅游其中，揣摩知识技巧，学习品味技术人生。

——上海宝信软件解决方案本部研究所所长兼技术总监 王森

应用开发和数据库优化我都做过，但是一直缺乏系统的指导。所以，我一直期待能有一本理论联系实际，透彻分析DB2性能优化思路并结合实际案例的书来提升技术水平。这本书大大超出了我的预期，它真称得上是分析和解决性能问题的神兵利器。

——亚信联创科技(中国)有限公司数据库资深顾问 郭占海

上架建议：计算机/数据库

随书案例请到www.tup.tsinghua.edu.cn网站下载

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks